This document has been approved for public release and sale: its distribution is unlimited

MIL-STD-1589C (USAF)
6 July 1984
SUPERSEDING
MIL-STD-1589B (USAF)
6 June 1980

MILITARY STANDARD

JOVIAL (J73)

No deliverable data is required by this document.

FSC IPSC

DEPARTMENT OF AIR FORCE

Washington DC 20301

JOVIAL (J73)

MIL-STD-1589C

1. This Military Standard is approved for use by the Department of the Air Force, and is available for use by all Departments and Agencies of the Department of Defense.

2. Recommendations, corrections, additions, or deletions should be addressed to: Commander, Rome Air Development Center, Attn: RBE-2, Griffiss AFB NY 13441.

FOREWORD

This revised document was coordinated as Change Notice 1 to MIL-STD-1589B. Due to the extent of changes it has been approved as MIL-STD-1589C. This version contains no changes from the coordinated version other than those required for reformatting. MIL-STD-1589C (USAF) defines the upgraded JOVIAL (J73) programming language.

The sections are organized in a top-down manner. The first section describes the interactions between the modules of the complete program so that in subsequent sections the structures of the language can be described (to the extent possible) without reference to their interaction with other structures.

Most sections are divided into separate parts entitled "Syntax," "Semantics," and "Constraints." The "Syntax" descriptions define the grammar of the language in a modified BNF notation. The "Semantics" discussions define the meaning of constructs that satisfy the Syntax and Constraints. The "Constraints" discussions enumerate non-syntactic requirements that must be met in order for the given constructs to be legal. The intent is that the Syntax, Semantics, and Constraints not be redundant with each other - e.g., the Semantics sections do not normally repeat something that should be obvious from the Syntax, neither do they repeat stipulations that are listed as Constraints.

Some of the designated Constraints apply at compile time, and others pertain to errors that are not detectable until the compiled program is executed. In order to conform to this standard, a J73 compiler must detect compile-time errors, but it is not required to generate code for run-time checks.

The Appendix provides a cross-reference index to constructs that appear in the Syntax. For each construct, the index gives the number of the section where that construct is defined and the numbers of the sections where that construct is used in a definition.

The following metalanguage conventions have been observed in this document:

1. Terminal symbols, i.e., those which actually appear in a program are written in upper case. For example:

```
BEGIN
END
STATIC
```

2. Non-terminal symbols, i.e., those which represent groups of terminal symbols are written in lower case and enclosed between < and >. If any non-terminal symbol is longer than one word, the words are separated by a hyphen. For example:

```
<compool-module>
<ordinary-table-body>
```

3. The following special symbols are used in the metalanguage:

::= means "is defined as." For example,

<a> ::= <b> <c>

where <a> is defined as the string <b> followed by the string <c>. Definitions that do not fit on one line may extend to the next line or lines.

| The | symbol indicates that what follows is an alternate choice of definition for the non-terminal to the left of the ::= symbol. For example,

<a> ::= <b> | <c>

where <a> is defined as either the string <b> or the string <c>.

[ ] If a string may optionally be present, it is enclosed between [and]. For example,

<a> ::= [<b>] <c>

where <a> is defined as either the string <c> or the string <b> followed by the string <c>.

4. The following symbols have metalinguistic meaning when appended to a non-terminal:

... One or more instances of the string represented by a non-terminal

,... One or more instances separated by a comma

:... One or more instances separated by a colon

For example:

<a>... Represents a single <a> or any sequence of <a>'s (e.g., <a> or <a> <a> or <a> <a> <a> <a> etc.)

[<a>...] Represents the null string or any sequence of <a>'s

<a>,... Represents a single <a> or any length sequence of <a>'s separated by commas (e.g., <a> or <a>,<a> or <a>,<a>,<a> etc.)

<a>:... Represents a single <a> or any length sequence of <a>'s separated by colons (e.g., <a> or <a>:<a> or <a>:<a>:<a> etc.)

5. If a non-terminal appearing on the right side of the ::= is not defined in that same sub-section, the number of the sub-section where it is defined appears in parentheses in the right margin.

6. In a "Semantics" or "Constraints" section, non-terminal symbols are enclosed between < and > when the usage refers to constructs occurring in a "syntax" section or when the specific J73 meaning might be confused with generalized programming usage.

Throughout this document, the symbols used for the prime, the quotation mark, and a blank are as follows:

1. Prime (')

2. Quotation mark (")

3. Blank ( )

Characters and symbols are defined in Section 8.0 of this document.

CONTENTS

Paragraph			Page
	1.0	GLOBAL CONCEPTS	1
	1.1	The complete program	1
	1.2	Modules	1
	1.2.1	Compool modules	1
	1.2.2	Procedure modules	2
	1.2.3	Main program modules	3
	1.3	Special considerations	3
	1.3.1	Conditional compilation	3
	1.3.2	Scope of names	4
	1.4	Implementation parameters	5
	2.0	DECLARATIONS	11
	2.1	Data declarations	12
	2.1.1	Item declarations	13
	2.1.1.1	Integer type descriptions	14
	2.1.1.2	Floating type descriptions	15
	2.1.1.3	Fixed type descriptions	16
	2.1.1.4	Bit type descriptions	18
	2.1.1.5	Character type descriptions	19
	2.1.1.6	Status type descriptions	19
	2.1.1.7	Pointer type descriptions	21
	2.1.2	Table declarations	22
	2.1.2.1	Table dimension lists	23
	2.1.2.2	Table structure	24
	2.1.2.3	Ordinary table entries	26
	2.1.2.4	Specified table entries	28
	2.1.3	Constant declarations	30
	2.1.4	Block declarations	31
	2.1.5	Allocation of data objects	32
	2.1.6	Initialization of data objects	33
	2.2	Type declarations	35
	2.3	Statement name declarations	38
	2.4	Define declarations	38
	2.4.1	Define calls	39
	2.5	External declarations	42
	2.5.1	DEF specifications	42
	2.5.2	REF specifications	43
	2.6	Overlay declarations	45
	2.7	Null declarations	46
	3.0	PROCEDURES AND FUNCTIONS	48
	3.1	Procedures	49
	3.2	Functions	50
	3.3	Parameters of procedures and functions	51

CONTENTS - Continued

Paragraph			Page
	3.4	Inline procedures and functions	55
	3.5	Machine-specific procedures and functions	56
	4.0	STATEMENTS	57
	4.1	Assignment statements	58
	4.2	Loop statements	59
	4.3	IF statements	62
	4.4	CASE statements	62
	4.5	Procedure call statements	65
	4.6	RETURN statements	66
	4.7	GOTO statements	66
	4.8	EXIT statements	67
	4.9	STOP statements	67
	4.10	ABORT statements	68
	5.0	FORMULAS	69
	5.1	Numeric formulas	71
	5.1.1	Integer formulas	71
	5.1.2	Floating formulas	73
	5.1.3	Fixed formulas	75
	5.2	Bit formulas	78
	5.2.1	Relational expressions	79
	5.2.2	Boolean formulas	81
	5.3	Character formulas	81
	5.4	Status formulas	82
	5.5	Pointer formulas	83
	5.6	Table formulas	83
	6.0	DATA REFERENCES	85
	6.1	Variable and block references	85
	6.2	Named constants	89
	6.3	Function calls	91
	6.3.1	LOC function	92
	6.3.2	NEXT function	93
	6.3.3	BIT function	94
	6.3.4	BYTE function	95
	6.3.5	Shift functions	95
	6.3.6	ABS function	96
	6.3.7	Sign function	96
	6.3.8	Size functions	96
	6.3.9	Bounds functions	98
	6.3.10	NWDSEN function	98

CONTENTS - Continued

			Page
Paragraph	6.3.11	Status inverse functions	99
	6.3.12	NENT function	99
	7.0	TYPE MATCHING AND TYPE CONVERSIONS	101
	8.0	BASIC ELEMENTS	109
	8.1	Characters	109
	8.2	Symbols	110
	8.2.1	Names	111
	8.2.2	Reserved words	112
	8.2.3	Operators	113
	8.2.4	Separators	114
	8.3	Literals	115
	8.3.1	Numeric literals	115
	8.3.2	Bit literals	117
	8.3.3	Boolean literals	118
	8.3.4	Character literals	118
	8.3.5	Pointer literals	120
	8.4	Comments	120
	8.5	Blanks	120
	9.0	DIRECTIVES	122
	9.1	Compool directives	123
	9.2	Text directives	124
	9.2.1	Copy directives	124
	9.2.2	Skip, begin, and end directives	125
	9.3	Linkage directives	125
	9.4	Trace directives	126
	9.5	Interference directives	127
	9.6	Reducible directives	128
	9.7	Listing directives	128
	9.7.1	Source-listing directives	128
	9.7.2	Define-listing directives	129
	9.8	Register directives	129
	9.9	Expression evaluation order directives	130
	9.10	Initialization directives	131
	9.11	Allocation order directives	131
Table	8-1	Bit-literal bead values	119
Appendix		Cross reference index	133

1.0 GLOBAL CONCEPTS

1.1 THE COMPLETE PROGRAM

Syntax:

<complete-program>	::=	<module>...	
<module>	::=	<compool-module>	(1.2.1)
	|	<procedure-module>	(1.2.2)
	|	<main-program-module>	(1.2.3)

Semantics:

A <complete-program> of the J73 language gives the complete specification of a computational algorithm to be performed. A <complete-program> consists of a group of one or more <modules> that are compilable separately and which may be subsequently bound together for execution as a unit. A <module> is the smallest entity in the language that may be separately compiled.

A <complete-program> may contain zero or more <compool-modules> and zero or more <procedure-modules>.

Constraint:

A <complete-program> must contain exactly one <main-program-module>.

Note:

A compiler may accept a file containing more than one <module>, but it is not required to do so. If it does accept such a file, it must process each <module> as though it had been submitted separately.

1.2 MODULES

1.2.1 COMPOOL MODULES

Syntax:

<compool-module>	::=	START COMPOOL <compool-name> ; [<compool-declaration>...] TERM	(2.0)
<compool-name>	::=	<name>	(8.2.1)

Semantics:

<Compool-modules> provide a means of declaring data objects, types, and subroutines that are to be made external - i.e., that are potentially available to other <modules> in the <complete-program>. Another <module> may access the names declared in a given <compool-module> by use of a <compool-directive> (see Section 9.1) that names the given compool or by use of external declarations (see Section 2.5).

A <compool-module> may contain <compool-directives> that name other <compool-modules>.

By appropriate use of <def-specifications> and <ref-specifications> within <compool-declarations>, a user can control whether physical allocation takes place within the <compool-module> itself or within the accessing <module> (see Section 2.5).

1.2.2 PROCEDURE MODULES

Syntax:

```
<procedure-module>          ::= START                                    (2.0)
                                [<declaration>...]
                                [<non-nested-subroutine>...]
                                TERM

<non-nested-subroutine>     ::= [DEF] <subroutine-definition>            (3.0)
```

Semantics:

<Procedure-modules> provide a means of separately compiling subroutines that specify portions of the actions of the <complete-program>.

If a <subroutine-definition> is preceded by DEF, that subroutine may be invoked from within the <main-program-module> or from within another <procedure-module>, provided that the referencing module contains an appropriate <ref-specification> for the subroutine or accesses a compool containing such a specification.

<Non-nested-subroutines> defined without a DEF may be invoked only from within the <procedure-module> or <main-program-module> in which they are defined. Similarly, all declarations in a <procedure-module> apply only within that <procedure-module> (unless they are <external-declarations> - see Section 2.5).

1.2.3 MAIN PROGRAM MODULES

Syntax:

```
<main-program-module>     ::= START
                              [<declaration>...]              (2.0)
                              PROGRAM
                              <program-name>  ;
                              <program-body>
                              [<non-nested-subroutine>...]    (1.2.2)
                              TERM

<program-name>            ::= <name>                          (8.2.1)

<program-body>            ::= <statement>                     (4.0)

                            | BEGIN [<declaration>...]        (2.0)
                              <statement>...                  (4.0)
                              [<subroutine-definition>...]    (3.0)
                              [<label>...] END                (4.0)
```

Semantics:

The body of a <main-program-module> is executed at the start of a <complete-program>. When execution of the body is complete, execution of the <complete-program> is complete. Unless the <complete-program> consists of a single <main-program-module>, the <main-program-module> will contain one or more <compool-directives>, references to externally-declared data, and/or calls of DEF'd subroutines in other modules.

Declarations in a <main-program-module> may be external or internal. If a <non-nested-subroutine> has a DEF, it may be invoked either locally or from within a <procedure-module>, provided that the referencing module contains an appropriate <ref-specification> for the subroutine or accesses a compool containing such a specification. If it does not have a DEF, it can be invoked only from within the module in which it is defined.

Constraints:

The <program-body> must contain at least one non-null statement (e.g., STOP).

1.3 SPECIAL CONSIDERATIONS

1.3.1 CONDITIONAL COMPILATION

Two methods are provided for conditionally suppressing generation of object code for portions of a JOVIAL module.

The !SKIP, !BEGIN, and !END directives (see Section 9.2.2) permit almost complete suppression of processing of suppressed source. The only processing

done for suppressed source is to scan for the terminating !END directive. Therefore the suppressed source may contain errors and/or statements incompatible with other module source without affecting compilation.

The IF and CASE statements (see Sections 4.3 and 4.4) permit suppression of generation of object code. Source for this suppressed object code must be correct since it is subject to the same validity checks and processing of directives as other source code. Only code that is unconditionally unreachable is suppressed so this conditional compilation must produce the same results as if the code was generated. Segments of code which are unreachable due to values of <if-statement> <boolean-formulas> or <case-selector-formulas> which are <compile-time-formulas> and which do not contain <labels> are always suppressed. Implementations may choose to do a more complete analysis and also suppress other recognized unreachable code.

1.3.2 SCOPE OF NAMES

<Procedure-modules> and the <main-program-module> can contain subroutines (i.e., procedures and functions) nested to any depth. Each subroutine, as well as the <program-body> and the <main-program-module> or <procedure-module> itself, establishes a region or scope for which a name's declaration is active and in which the <name> can be used. The scope of a <name> is that region of the <complete-program> within which that <name> has a single meaning.

A name declared with a DEF or REF (see Section 2.5) is considered to be external; all other names are internal. An external <name> can be used in any module of the <complete-program>, except within a scope containing an internal name with the same spelling. An internal name can be used only within the subroutine, <procedure-module>, or <main-program-module> within which that name is declared, but not within an enclosed scope containing a <name> with the same spelling.

The <name> of a subroutine belongs to the scope in which that subroutine is declared or defined.

For any given compilation, all names made available from referenced <compool-modules> (see Section 9.1), as well as the name of the <module> being compiled and all <compool-names>, belong to the same scope, referred to as compool scope, which is considered to enclose the scope established by the <procedure-module>, <main-program-module>, or <compool-module> being compiled.

System-defined names (e.g., machine-specific subroutines, implementation parameters) belong to system scope, which encloses the compool scope. Such names may be redefined by the programmer.

These rules ensure that any two names with the same spelling but with distinct scopes are regarded as if they were different names.

Constraints:

No two names having the same scope may have the same spelling. (This constraint does not prevent two tables with different <table-names> to be declared in the same scope using the same <table-type-name>. See Sections 2.1.2 and 2.2.)

No two external names may have the same spelling.

1.4 IMPLEMENTATION PARAMETERS

Syntax:

```
<integer-machine-
    parameter>      ::=  BITSINBYTE

                       ¦  BITSINWORD

                       ¦  LOCSINWORD

                       ¦  BYTEPOS
                          (  <compile-time-integer-formula>  )  (5.1.1)

                       ¦  BYTESINWORD

                       ¦  BITSINPOINTER

                       ¦  INTPRECISION

                       ¦  FLOATPRECISION

                       ¦  FIXEDPRECISION

                       ¦  FLOATRADIX

                       ¦  IMPLFLOATPRECISION
                          (  <precision>  )                  (2.1.1.2)

                       ¦  IMPLFIXEDPRECISION
                          (  <scale-specifier>  ,            (2.1.1.3)
                          <fraction-specifier  )             (2.1.1.3)

                       ¦  IMPLINTSIZE
                          (  <integer-size>  )               (2.1.1.1)

                       ¦  MAXFLOATPRECISION

                       ¦  MAXFIXEDPRECISION

                       ¦  MAXINTSIZE
```

```
                    ¦ MAXBYTES

                    ¦ MAXBITS

                    ¦ MAXINT ( <integer-size> )          (2.1.1.1)

                    ¦ MININT ( <integer-size> )          (2.1.1.1)

                    ¦ MAXTABLESIZE

                    ¦ MAXSTOP

                    ¦ MINSTOP

                    ¦ MAXSIGDIGITS

                    ¦ MINSIZE (
                      <compile-time-integer-formula> )   (5.1.1)

                    ¦ MINFRACTION (
                      <compile-time-floating-formula> )  (5.1.2)

                    ¦ MINSCALE (
                      <compile-time-floating-formula> )  (5.1.2)

                    ¦ MINRELPRECISION (
                      <compile-time-floating-formula )   (5.1.2)

<floating-machine-
    parameter>  ::= MAXFLOAT ( <precision> )             (2.1.1.2)

                    ¦ MINFLOAT ( <precision> )           (2.1.1.2)

                    ¦ FLOATRELPRECISION
                      ( <precision> )                    (2.1.1.2)

                    ¦ FLOATUNDERFLOW
                      ( <precision> )                    (2.1.1.2)

<fixed-machine-
    parameter>  ::= MAXFIXED ( <scale-specifier> ,       (2.1.1.3)
                      <fraction-specifier> )             (2.1.1.3)

                    ¦ MINFIXED ( <scale-specifier> ,     (2.1.1.3)
                      <fraction-specifier> )             (2.1.1.3)
```

Semantics:

The machine on which a J73 program runs contains an array of memory cells. These cells are grouped or partitioned into the following units for purposes of the language specification.

1. Bit - The smallest unit of storage (can contain one of two values, which are represented by zero and one)

2. Byte - A group of one or more consecutive bits that is capable of holding a single character of information

3. Word - A memory partition of one or more consecutive bits that serves as the unit of allocation of data storage

4. Address Unit - The machine dependent unit used to identify an address or location in memory

The number of bits per byte, word, and address varies from implementation to implementation, and these quantities affect the representation and behavior of data in the language. Machine parameters are constants that describe these implementation-dependent differences. The values of these constants must be specified as part of the implementation of a J73 compiler on any computer. These names can then be referenced by a user to access the values associated with that implementation.

The size of an <integer-machine-parameter> is the size of an <integer-literal> having that value. The attributes of a <floating-machine-parameter> or <fixed-machine-parameter> are as specified by its <precision> or its <scale-specifier> and <fraction-specifier>. The values of the implementation parameters are as follows:

BITSINBYTE	Number of bits in a byte
BITSINWORD	Number of bits in a word
LOCSINWORD	Number of locations (address units) in a word
BYTEPOS(PP)	A permitted <starting-bit> value for character strings that cross word boundaries. PP is any integer value between 0 and BYTESINWORD-1, inclusive and BYTEPOS(PP) < BYTEPOS(PP+1).
BYTESINWORD	Number of complete bytes in a word
BITSINPOINTER	Number of bits used for a pointer value
INTPRECISION	The number of bits that an implementation supplies to hold the value of an integer item (exclusive of sign, if any) when no <integer-size> is specified by the programmer.
FLOATPRECISION	The number of bits that an implementation supplies to hold the value of the mantissa of a floating point item (exclusive of the sign bit) when no <precision> is specified by the programmer

FIXEDPRECISION	The number of bits that an implementation supplies to hold the value of a fixed item (exclusive of the sign bit) when no <fraction-specifier> is supplied by the programmer
FLOATRADIX	Base of the floating point representation, specified as an integer
IMPLFLOATPRECISION(II)	Number of bits (not including the sign bit) in the mantissa of the representation for a floating point value whose specified precision is II
IMPLFIXEDPRECISION(SS,FF)	The number of bits (excluding sign bit) an implementation uses to represent an unpacked fixed item with scale SS and fraction FF. This value also determines the accuracy of fixed formula results.
IMPLINTSIZE(II)	The number of bits (excluding sign bit) an implementation uses to represent an unpacked S or U item with specified size II.
MAXFLOATPRECISION	Maximum specifiable precision supported by an implementation for a <floating-item-description>
MAXFIXEDPRECISION	Maximum value supported by an implementation for the sum of the scale and fraction specifiers in a <fixed-item-description>
MAXINTSIZE	Maximum specifiable size (not including the sign bit) supported by an implementation for signed and unsigned integers
MAXBYTES	Maximum value supported by an implementation for a <character-size>; must not exceed MAXBITS/BITSINBYTE
MAXBITS	Maximum value supported by an implementation for a <bit-size>; the maximum value of words per entry in a table is MAXBITS/BITSINWORD, and the maximum BITSIZE of a table entry is MAXBITS
MAXINT(SS)	Maximum integer value representable in SS+1 bits (including sign bit)
MININT(SS)	Minimum signed integer value representable in SS+1 bits (including sign bit), using the implementation's method of representing negative numbers

MAXTABLESIZE	The maximum number of words an implementation permits a table to occupy.
MAXSTOP	Maximum specifiable value for an <integer-formula> in a <stop-statement> (see Section 4.9)
MINSTOP	Minimum specifiable value for an <integer-formula> in a <stop-statement> (see Section 4.9)
MAXSIGDIGITS	The maximum number of significant digits an implementation will process for a fixed or floating point literal (see Section 8.3.1)
MINSIZE(II)	The minimum value of SS such that II is less than or equal to MAXINT(SS) and greater than or equal to MININT(SS)
MINFRACTION(AA)	The minimum value of FF such that 2**(-FF) is less than or equal to AA. The value of AA must be greater than zero.
MINSCALE(AA)	The minimum value of SS such that 2**SS is greater than AA. The value of AA must be greater than zero.
MINRELPRECISION(FF)	The minimum value of PP such that FLOATRELPRECISION(PP) is less than or equal to FF. The value of FF must be greater than or equal to FLOATRELPRECISION (MAXFLOATPRECISION).
MAXFLOAT(PP)	Maximum floating point value using only the first PP mantissa bits (excluding sign) of the implementation's floating point representation whose actual mantissa length is IMPLFLOATPRECISION(PP). PP must be greater than zero and not exceed MAXFLOATPRECISION.
MINFLOAT(PP)	Minimum floating point value representable in exactly PP mantissa bits, (excluding sign) and using the implementation's method of representing negative numbers. PP must be greater than zero and not exceed MAXFLOATPRECISION.
FLOATRELPRECISION(PP)	Let FRP1 be the smallest floating point value greater than 1.0 using the first PP bits (excluding sign) of the implementation's representation for floating point values. FLOATRELPRECISION(PP) equals FRP1 - 1.0. PP

	must be greater than zero and not exceed MAXFLOATPRECISION.
FLOATUNDERFLOW(PP)	The smallest positive floating point value using exactly PP mantissa bits (excluding sign) and such that both FLOATUNDERFLOW(PP) and -FLOATUNDERFLOW(PP) are representable as floating point values
MAXFIXED(SS,FF)	Maximum fixed value representable in SS+FF+1 bits (including sign bit)
MINFIXED(SS,FF)	Minimum fixed value representable in SS+FF+1 bits (including sign bit), using the implementation's method for representing negative values

Note:

A FIXEDRADIX implementation parameter is not provided since fixed point values are represented using radix 2 (see Section 2.1.1.3).

2.0 DECLARATIONS

Syntax:

<declaration>	::=	<data-declaration>	(2.1)
	|	<type-declaration>	(2.2)
	|	<subroutine-declaration>	(3.0)
	|	<statement-name-declaration>	(2.3)
	|	<define-declaration>	(2.4)
	|	<external-declaration>	(2.5)
	|	<overlay-declaration>	(2.6)
	|	<inline-declaration>	(3.4)
	|	<null-declaration>	(2.7)
	|	BEGIN <declaration>... END	

<compool-declaration>	::=	<external-declaration>	(2.5)
	|	<constant-declaration>	(2.1.3)
	|	<type-declaration>	(2.2)
	|	<define-declaration>	(2.4)
	|	<overlay-declaration>	(2.6)
	|	<null-declaration>	(2.7)
	|	BEGIN <compool-declaration>... END	

Semantics:

<Declarations> associate <names> with programmer-supplied meanings.

A <compool-declaration> is a <declaration> that appears in a <compool-module>.

Constraints

Except for <statement-names>, names of subroutines, type names in <pointer-item-descriptions>, and formal parameter names in subroutine <formal-parameter-lists>, a name may not be used prior to the point at which a <declaration> for that name appears.

2.1 DATA DECLARATIONS

Syntax:

<data-declaration>	::=	<item-declaration>	(2.1.1)
	|	<table-declaration>	(2.1.2)
	|	<constant-declaration>	(2.1.3)
	|	<block-declaration>	(2.1.4)

Semantics:

<Data-declarations> declare <data-names> and their attributes. Three kinds of data structures exist in J73:

1. Item - A simple data object of the language. An item is a variable of a pre-defined or programmer-defined type having no constituents.

2. Table - An aggregate data object consisting of a collection of one or more items, or an array of such collections. The collection of items is called an entry. An entire entry in a table is selected by the use of the table name, together with a sequence of indices ("subscripts") if the table is arrayed. An item within an entry is selected by the use of the item name and the appropriate number of subscripts.

3. Block - A group of items and tables and other blocks to which is allocated a contiguous area of storage.

Additionally, an item or table may be declared to be CONSTANT, in which case its value cannot be changed during execution. A constant item must be given an initial value by means of an <item-preset>. Blocks, items, or tables (other than constants) can specify, by means of an <allocation-specifier>, the allocation permanence of the storage associated with their names. Non-constant items and tables can optionally be given initial values by means of <item-presets> or <table-presets>.

The value of an uninitialized data object is undefined until it receives a value in an executable statement.

Declarations associate a <name> with a type. A type determines the set of values that an object can have and the operations that can be performed on those values. Types are grouped into related sets called type classes. The type

classes are integer, floating, fixed, bit, character, status, pointer, and table. Types within a type class are distinguished by the values of certain properties known as attributes. Rules concerning type matching are found in Section 7.0.

2.1.1 ITEM DECLARATIONS

Syntax:

```
<item-declaration>        ::= ITEM <item-name>
                              [<allocation-specifier>]        (2.1.5)
                              <item-type-description>
                              [<item-preset>] ;               (2.1.6)

<item-name>               ::= <name>                          (8.2.1)

<item-type-description>   ::= <integer-type-description>      (2.1.1.1)
                            | <floating-type-description>     (2.1.1.2)
                            | <fixed-type-description>        (2.1.1.3)
                            | <bit-type-description>          (2.1.1.4)
                            | <character-type-description>    (2.1.1.5)
                            | <status-type-description>       (2.1.1.6)
                            | <pointer-type-description>      (2.1.1.7)
```

Semantics:

<Item-declarations> declare items. Items are used as variables to retain values in a J73 program. Allocation for items declared in <item-declarations> will be such that no items share a word.

The <item-type-description> establishes the type of an item.

The <allocation-specifier> establishes the allocation permanence of items which are not enclosed in blocks. This allocation permanence is automatic if the declaration is in a subroutine and the <allocation-specifier> is omitted, otherwise it is STATIC (see Section 2.1.5). Items enclosed in blocks inherit the allocation permanence of the enclosing block.

The <item-preset>, if present, specifies an initial value for the item.

Constraints:

Only items having STATIC allocation (explicitly or by default) may contain an <item-preset>.

Declarations of items that are <formal-input-parameters> or <formal-output-parameters> (see Section 3.3) must not contain an <allocation-specifier> or <item-preset>.

An <item-declaration> within a block must not contain an <allocation-specifier>.

2.1.1.1 INTEGER TYPE DESCRIPTIONS

Syntax:

<integer-type-description>	::=	<integer-item-description>	
	¦	<integer-type-name>	
<integer-item-description>	::=	S [<round-or-truncate>] [<integer-size>]	(2.1.1.2)
	¦	U [<round-or-truncate>] [<integer-size>]	(2.1.1.2)
<integer-size>	::=	<compile-time-integer-formula>	(5.1.1)
<integer-type-name>	::=	<item-type-name>	(2.2)

Semantics:

An <integer-type-description> is used to specify a signed integer type or an unsigned integer type. S specifies a signed integer type; U specifies an unsigned integer type.

The <integer-size> attribute specifies the minimum number of bits of storage required to hold the maximum value of the integer (excluding the sign, if any). If <integer-size> is omitted, it defaults to INTPRECISION. The number of bits allocated for signed integers will be at least <integer-size>+1, and for unsigned integers will be at least <integer-size>.

The value set for a signed integer type with size SS is MININT(SS) through MAXINT(SS). The value set for an unsigned integer type with size SS is 0 through MAXINT(SS).

The <round-or-truncate> attribute specifies truncation or rounding is to occur when a value is converted to an integer type. If R is specified, rounding will occur. If T is specified, truncation towards minus infinity will occur. If Z is specified, truncation towards zero will occur. If the attribute is omitted, truncation in an implementation-dependent manner will occur.

Constraints:

The maximum value that can be specified for <integer-size> is MAXINTSIZE, an implementation parameter.

<Integer-size> must be greater than zero.

An <integer-type-name> must be an <item-type-name> declared in an <item-type-declaration> that contains an <integer-type-description> (see Section 2.2).

Notes:

An implementation may choose MAXINTSIZE > BITSINWORD-1.

The <round-or-truncate> option has a use only when an <integer-item-description> is used in an <integer-conversion> (see Section 7.0).

2.1.1.2 FLOATING TYPE DESCRIPTIONS

Syntax:

```
<floating-type-description> ::= <floating-item-description>
                              | <floating-type-name>

<floating-item-description> ::= F [<round-or-truncate>]
                                [<precision>]

<round-or-truncate>         ::= , R
                              | , T
                              | , Z

<precision>                 ::= <compile-time-integer-formula>(5.1.1)

<floating-type-name>        ::= <item-type-name>              (2.2)
```

Semantics:

A <floating-type-description> is used to specify a floating type. The <precision> attribute specifies the minimum number of bits of storage required to hold the value of the mantissa. If <precision> is omitted, it defaults to FLOATPRECISION, an implementation parameter.

The <round-or-truncate> attribute is used to specify whether truncation or rounding is to occur when a value of a floating type with a greater <precision> is assigned to an item of this type. If R is specified, rounding will occur. If T is specified, truncation towards minus infinity will occur. If Z is specified, truncation towards zero will occur. If the attribute is omitted, truncation in an implementation-dependent manner will occur. Rounding and

truncation take place with respect to the implemented precision of the floating type. (Note: IMPLFLOATPRECISION(PP) is an implementation parameter defining what precision is provided when precision PP is specified.)

The value set for a floating type with <precision> PP is MINFLOAT(PP) through -FLOATUNDERFLOW(PP), 0, and FLOATUNDERFLOW(PP) through MAXFLOAT(PP).

Constraints:

The maximum value that can be specified for <precision> is MAXFLOATPRECISION, an implementation parameter.

<Precision> must be greater than zero.

A <floating-type-name> must be an <item-type-name> declared in an <item-type-declaration> that contains a <floating-type-description> (see Section 2.2).

Note:

Since a <floating-type-description> specifies only the minimum precision required, an implementation is free to support only one or two levels of implemented precision. Which implemented precision level represents a floating type depends on the value of the specified precision. The implemented precision must never be less than the specified precision. Since an implementation may provide more than the specified precision, it is consistent to round or truncate a represented value only if converting from a longer to a shorter implemented precision.

2.1.1.3 FIXED TYPE DESCRIPTIONS

Syntax:

```
<fixed-type-description>  ::=  <fixed-item-description>
                           |   <fixed-type-name>

<fixed-item-description>  ::=  A [<round-or-truncate>]        (2.1.1.2)
                               <scale-specifier>
                               [, <fraction-specifier>]

<scale-specifier>         ::=  <compile-time-integer-formula>(5.1.1)

<fraction-specifier>      ::=  <compile-time-integer-formula>(5.1.1)

<fixed-type-name>         ::=  <item-type-name>               (2.2)
```

Semantics:

A <fixed-type-description> is used to specify a fixed point numeric type. If SS is the value of the <scale-specifier> and FF is the value of the <fraction-specifier>, then SS+FF is the minimum number of bits in the

representation, excluding the sign bit. When SS and FF are both positive, SS specifies the number of bits to the left of the binary point (excluding the sign bit) and FF the minimum number of bits to the right (see Note below). When SS is negative, the binary point is assumed to be ABS(SS) bits to the left of the first (non-sign) bit of the representation. Similarly, when FF is negative, the least significant bit of the representation is no more than ABS(FF) bits to the left of the binary point.

The (nominal) precision of a fixed point type is the sum of its scale and fraction specifier. The implemented precision may be greater than the nominal bits required. If <fraction-specifier> is omitted, the fixed point type has a default precision given by FIXEDPRECISION, an implementation parameter, and the implied value of the omitted <fraction-specifier> is FIXEDPRECISION-SS, where SS is the <scale-specifier>.

If FF is a fixed point item declared with a default <fraction-specifier>, then FIXEDPRECISION = BITSIZE(REP(FF))-1.

The <round-or-truncate> attribute specifies truncation or rounding is to occur when a value is converted to a fixed point type. If R is specified, rounding will occur. If T is specified, truncation towards minus infinity will occur. If Z is specified, truncation towards zero will occur. If the attribute is omitted, truncation in an implementation-dependent manner will occur. Rounding and truncation take place with respect to the implemented precision of the fixed type (see Note below).

The value set of a fixed point type with scale SS and fraction FF is MINFIXED(SS,FF) through MAXFIXED(SS,FF).

Constraints:

The sum of the scale and fraction specifiers (i.e., the nominal precision) must be greater than zero and must not exceed MAXFIXEDPRECISION, an implementation parameter.

The value of <scale-specifier> must lie in the range -127 through +127.

A <fixed-type-name> must be an <item-type-name> declared in an <item-type-declaration> that contains a <fixed-type-description> (see Section 2.2).

Notes:

The set of exactly representable fixed point values is determined by a fixed type's scale and fraction specifiers. A <fraction-specifier> value, FF, means fixed point values must be represented with a precision greater than or equal to 2**(-FF). A <scale-specifier> value, SS, means the maximum representable value is at least 2**SS - 2**(-FF) and less than 2**SS.

An implementation is permitted to support more than one level of implemented precision for fixed point types. For computational purposes, values will be represented using the smallest implemented precision level (e.g., one word or two words) consistent with the value's nominal precision. For storage purposes

in packed tables, a fixed point value need occupy no more than the number of bits specified by the nominal precision plus one bit for the sign.

IMPLFIXEDPRECISION(SS,FF) is an implementation parameter defining what precision is provided for an unpacked fixed point item when nominal precision SS+FF is specified. In addition, the implemented precision of a packed item (i.e., an item in a specified table, packed ordinary table, or a tight table) as well as an unpacked item is given by BITSIZE(REP(FI))-1, where FI is the fixed point item.

The implemented precision of a fixed item is the number of bits (excluding sign bit) used to store the item. Assignments to such items round or truncate with respect to this precision, which is never less than the specified precision. Rounding or truncation can change a fixed point value only if the implemented precision is shortened.

It should be noted that specifying R, T, or Z in an item declaration only affects the conversion of literal values (see Section 8.3.1) and assignments of fixed point values when the stored representation of the value is shorter than the representation used for computations.

2.1.1.4 BIT TYPE DESCRIPTIONS

Syntax:

<bit-type-description>	::=	<bit-item-description>	
	¦	<bit-type-name>	
<bit-item-description>	::=	B [<bit-size>]	
<bit-size>	::=	<compile-time-integer-formula>	(5.1.1)
<bit-type-name>	::=	<item-type-name>	(2.2)

Semantics:

A <bit-type-description> is used to specify a bit string type. The <bit-size> attribute specifies the number of bits in the string. If <bit-size> is omitted it defaults to 1.

Constraints:

The maximum value that can be specified for <bit-size> is MAXBITS, an implementation parameter. The minimum value that can be specified for <bit-size> is one.

A <bit-type-name> must be an <item-type-name> declared in an <item-type-declaration> that contains a <bit-type-description> (see Section 2.2).

2.1.1.5 CHARACTER TYPE DESCRIPTIONS

Syntax:

```
<character-type-description> ::= <character-item-description>
                               ¦ <character-type-name>

<character-item-description> ::= C [<character-size>]

<character-size>             ::= <compile-time-integer-formula>(5.1.1)

<character-type-name>        ::= <item-type-name>              (2.2)
```

Semantics:

A <character-type-description> is used to specify a fixed-length character string type. The <character-size> attribute specifies the number of characters in the string. If <character-size> is omitted it defaults to 1.

Constraints:

The maximum value that can be specified for <character-size> is MAXBYTES, an implementation parameter. The minimum value that can be specified for <character-size> is one.

A <character-type-name> must be an <item-type-name> declared in an <item-type-declaration> that contains a <character-type-description> (see Section 2.2).

2.1.1.6 STATUS TYPE DESCRIPTIONS

Syntax:

```
<status-type-description>  ::= <status-item-description>
                             ¦ <status-type-name>

<status-item-description>  ::= STATUS [<status-size>]
                               ( <status-list> )

<status-list>              ::= <default-sublist>
                             ¦ [<default-sublist> ,]
                               <specified-sublist>,...

<default-sublist>          ::= <status-constant>,...

<specified-sublist>        ::= <status-list-index>
                               <status-constant>,...
```

<status-list-index>	::=	<compile-time-integer-formula>	(5.1.1)
<status-constant>	::=	V (<status>)	
<status>	::=	<name>	(8.2.1)
	|	<letter>	(8.1)
	|	<reserved-word>	(8.2.2)
<status-type-name>	::=	<item-type-name>	(2.2)
<status-size>	::=	<compile-time-integer-formula>	(5.1.1)

Semantics:

A <status-type-description> is used to specify a status type. The <status-list> is used to define the value set of the type, which consists of a set of named <status-constants>. These named <status-constants> are considered to be the logical values of the status type. Associated with each logical value is a representational value, i.e., how the value is actually represented internally. If the <status-list> contains only a <default-sublist>, the status type is said to have a default representation. The <status-constants> in the <default-sublist> will be assigned representational values 0 through N-1 (where N is the number of <status-constants> in the sublist) in the order in which they are specified in the list. The <status-constants> in each <specified-sublist> will be assigned representational values <status-list-index> through <status-list-index> + N-1 (where N is the number of <status-constants> in the sublist) in the order in which they are specified.

For a given <status-list>, the value of any <status-constant> is considered to be greater than the value of another <status-constant> having a lower representational value.

<Status-size> specifies the minimum number of bits to be allocated to hold the status value (excluding the sign bit, if any). If it is omitted, it defaults to the minimum needed for the representation as an integer value. If the representation of the lowest-valued <status-constant> in the list is less than zero, signed integer representation will be used; otherwise, unsigned integer representation will be used.

Constraints:

The <status-constants> must be unique within the <status-list>.

The <status-list-indices> within a <status-list> must be specified such that all the <status-constants> in the <status-list> receive unique representational values.

The value specified in <status-size> must be greater than or equal to the minimum needed for the representation of the status values (excluding the sign bit, if any) and less than or equal to MAXINTSIZE.

The representation of a status value cannot be less than MININT (BITSINWORD-1) and it cannot exceed MAXINT(BITSINWORD-1).

A <status-type-name> must be an <item-type-name> declared in an <item-type-declaration> that contains a <status-type-description> (see Section 2.2).

Note:

The use of a <name> in a <status> does not constitute a declaration of the <name> or a reference to a declared <name> having the same spelling. Within a given scope, a <status> <name> and a declared <name> can have the same spelling and no conflict will result.

2.1.1.7 POINTER TYPE DESCRIPTIONS

Syntax:

```
<pointer-type-description>    ::=  <pointer-item-description>
                                 |  <pointer-type-name>

<pointer-item-description>    ::=  P  [<type-name>]

<pointer-type-name>           ::=  <item-type-name>            (2.2)

<type-name>                   ::=  <item-type-name>            (2.2)
                                 |  <table-type-name>           (2.2)
                                 |  <block-type-name>           (2.2)
```

Semantics:

A <pointer-type-description> is used to specify a pointer type. If the <pointer-item-description> contains a <type-name>, then the pointer being specified is a typed pointer. If the <type-name> is omitted, then the pointer is an untyped pointer.

A typed pointer contains the address of a data object of the type specified by the <type-name>. The object being pointed to may be obtained by dereferencing the pointer (see Section 6.1).

An untyped pointer contains the address of a data object of any type. However, such a pointer must be converted to a typed pointer (see Section 7.0) before it may be dereferenced or assigned to a typed pointer.

Constraint:

A <pointer-type-name> must be an <item-type-name> declared in an <item-type-declaration> that contains a <pointer-type-description> (see Section 2.2).

2.1.2 TABLE DECLARATIONS

Syntax:

<table-declaration>	::=	TABLE <table-name>	
		[<allocation-specifier>]	(2.1.5)
		[<dimension-list>]	(2.1.2.1)
		<table-description>	
<table-description>	::=	[<structure-specifier>]	(2.1.2.2)
		<entry-specifier>	
	|	<table-type-name>	(2.2)
		[<table-preset>] ;	(2.1.6)
<entry-specifier>	::=	<ordinary-entry-specifier>	(2.1.2.3)
	|	<specified-entry-specifier>	(2.1.2.4)
<table-name>	::=	<name>	(8.2.1)

Semantics:

<Table-declarations> declare named aggregate data objects. The presence of a <dimension-list> indicates that the table is an arrayed collection of entries. The <dimension-list> specifies the range of indices of the array.

The <allocation-specifier> establishes the allocation permanence of tables which are not enclosed in blocks. This allocation permanence is automatic if the declaration is in a subroutine and the <allocation-specifier> is omitted, otherwise it is STATIC (see Section 2.1.5). Tables enclosed in blocks inherit the allocation permanence of the enclosing block.

The <table-description> describes the contents of the table either with a <table-type-name> (see Section 2.2) or with an <entry-specifier>. Two or more tables may be declared in the same scope using the same <table-type-name>, and no name conflicts of the contained items will result, provided the <table-names> are different.

A table may either be an ordinary table, in which only the logical structure is described (see Section 2.1.2.3) or a specified table, in which the detailed physical layout of the table is described (see Section 2.1.2.4).

A <structure-specifier> is used to specify the representation of entries in a dimensioned table (see Section 2.1.2.2).

The <table-preset>, if present, specifies initial values for the table components. For <table-descriptions> containing an <entry-specifier> rather than a <table-type-name>, the <table-preset> is part of the <entry-specifier> (see Section 2.1.2.3 and 2.1.2.4).

Constraints:

Only tables having STATIC allocation (explicitly or by default) may contain a <table-preset>.

Tables that are <formal-input-parameters> or <formal-output-parameters> (see Section 3.3) must not contain an <allocation-specifier> or <table-preset>.

A <table-declaration> within a block must not contain an <allocation-specifier>.

If a <table-declaration> contains a <dimension-list> and its <table-description> consists of a <table-type-name>, the referenced table type must not be dimensioned.

A <structure-specifier> in an undimensioned table is prohibited.

Items in tables declared with a <table-type-name> can be accessed only by using pointers.

2.1.2.1 TABLE DIMENSION LISTS

Syntax:

```
<dimension-list>        ::=  (  <dimension>,...  )

<dimension>             ::=  [<lower-bound-option>]
                             <upper-bound>

                          |  *

<lower-bound-option>    ::=  <lower-bound>  :

<lower-bound>           ::=  <compile-time-integer-formula>   (5.1.1)

                          |  <compile-time-status-formula>    (5.4)

<upper-bound>           ::=  <compile-time-integer-formula>   (5.1.1)

                          |  <compile-time-status-formula>    (5.4)
```

Semantics:

A <dimension-list> specifies that a table is an array. Each <dimension> specifies the range of values for that dimension. If the <lower-bound> is omitted, it defaults to zero if the <upper-bound> is an integer; if the <upper-bound> is a status value, it defaults to the first <status-constant> in the status type of the <upper-bound>.

A <dimension> of * that appears with a <formal-input-parameter> or <formal-output-parameter> means the bounds will be determined from the actual parameter on each invocation. (Note that in accordance with Sections 6.3.9 and 6.1, bounds of * dimensions range from 0 to NN-1, where NN is the number of elements in the corresponding dimension of the actual parameter, regardless of what the lower and upper bounds values are for the actual parameter or whether the bound has an integer or status type.)

The type of a <dimension> with a status <upper-bound> is the type of the <upper-bound>. Otherwise the type is S IMPLINTSIZE(MINSIZE(maxbound)) where maxbound is MAXTABLESIZE for * dimensions and the bound with the greatest magnitude for numeric bounds.

Constraints:

Only status types with default representations may be used in <dimensions>.

The <lower-bound> must be less than or equal to the <upper-bound>.

The <lower-bound> and <upper-bound> must both be status formulas of the same type or both be integer formulas.

The maximum number of <dimensions> is seven.

A <dimension> of * may appear only in the <declaration> of a table <formal-input-parameter> or <formal-output-parameter>.

If any <dimension> of a table <formal-input-parameter> or <formal-output-parameter> is specified as *, they all must be specified as *.

The number of words occupied by a table must not exceed MAXTABLESIZE.

If either bound is a <status-constant> whose type is ambiguous (i.e. the <status-constant> belongs to more than one status type and is not explicitly disambiguated by a <status-conversion> -- see Section 7.0), then the other bound must have an independently determinable status type.

2.1.2.2 TABLE STRUCTURE

Syntax:

```
<structure-specifier>   ::=  PARALLEL
                           |  T  [<bits-per-entry>]

<bits-per-entry>        ::=  <compile-time-integer-formula>  (5.1.1)
```

Semantics:

Dimensioned tables can have a parallel or serial structure. In addition, a serial table may be tightly structured. The <structure-specifier> specifies the table structure.

A <structure-specifier> PARALLEL indicates parallel structure. For tables with parallel structure, the first word (word 0) of each entry is allocated consecutively, then word one, etc. An omitted <structure-specifier> or one with T indicates serial structure. For tables with a serial structure, all words of the first entry are allocated consecutively, then all words of the next entry, etc. Entries in both parallel and serial tables are arranged such that the rightmost indices vary fastest, from the lower bound to the upper bound.

A <structure-specifier> of T indicates tight structure (in addition to serial structure). Tight structure defines the allocation of storage between entries in a dimensioned (ordinary or specified) table, whereas packing (see Section 2.1.2.3) defines the allocation of storage within an entry of an ordinary table. Tight structure indicates that multiple entries of a dimensioned table are to be stored within a single word such that no entry crosses a word boundary. <Bits-per-entry> specifies the number of bits each entry is to occupy. If it is omitted, it will default to the minimum number of bits needed to store the entry.

Entries in tightly-structured tables are right-justified in the bits allotted. Items in an entry of an ordinary tight-structured table are right-justified within the bits allotted for the entry.

Entries in tables without a <structure-specifier> of T shall not share a word.

Constraints:

<Bits-per-entry> must be equal to or greater than the minimum number of bits needed to store the entry.

The <compile-time-integer-formula> must not be an <integer-primary> of the form <integer-conversion> (<formula>).

The explicit or default value of <bits-per-entry> must be less than or equal to BITSINWORD.

Items in a parallel table must not cross word boundaries.

2.1.2.3 ORDINARY TABLE ENTRIES

Syntax:

```
<ordinary-entry-
   specifier>            ::= [<packing-specifier>]
                             <item-type-description>          (2.1.1)
                             [<table-preset>] ;               (2.1.6)

                           | [<packing-specifier>]
                             [<table-preset>] ;               (2.1.6)
                             <ordinary-table-body>

<packing-specifier>      ::= N

                           | M

                           | D

<ordinary-table-body>    ::= <ordinary-table-item-declaration>

                           | BEGIN
                             <ordinary-table-options>...
                             END

<ordinary-table-item-
   declaration>          ::= ITEM <table-item-name>
                             <item-type-description>          (2.1.1)
                             [<packing-specifier>]
                             [<table-preset>] ;               (2.1.6)

<table-item-name>        ::= <name>                           (8.2.1)

<ordinary-table-options> ::= <ordinary-table-item-declaration>


                           | <null-declaration>               (2.7)
```

Semantics:

An <ordinary-entry-specifier> is used to specify the contents of an entry of an ordinary table.

No allocation order is implied by the order of items in the <ordinary-table-body> unless the <ordinary-table-body> contains an <order-directive>. If an <order-directive> is not in effect, <ordinary-table-options> will be reordered, if necessary, to reduce the storage occupied by an entry, consistent with the <packing-specifier>. Tables having the same type will have the same representation.

The <packing-specifier> specifies the density with which items are allocated within an entry. The following three degrees of packing can be specified:

1. N indicates that the items are not packed. No items share a word.

2. M indicates a density of packing that can be between N and D. The exact meaning is implementation-dependent and is specified to be an effective compromise between space usage and accessing ease.

3. D indicates dense packing. Items are allocated adjacent bits in a word with the following exceptions:

    a. Non-character items one word or longer start on a word boundary. Shorter non-character items do not cross word boundaries.

    b. Each byte of a character item which crosses a word boundary must be allocated on a byte boundary. An implementation may (but need not) allocate the bytes of other character items on byte boundaries.

A <packing-specifier> preceding an <ordinary-table-body> in an <ordinary-entry-specifier> applies to all items in the <ordinary-table-body> that do not themselves include a <packing-specifier> in their declaration.

Default packing for a tightly-structured table (see Section 2.1.2.2) is D; for all other tables, it is N.

The value of unallocated bits in an <ordinary-entry-specifier> is implementation-dependent.

The <table-preset>, if present, specifies initial values for the table entries.

Constraints:

Only tables having STATIC allocation (implicitly or explicitly) may contain a <table-preset>.

Declaration of table <formal-input-parameters> or <formal-output-parameters> (see Section 3.3) must not contain <table-presets>.

If a <table-preset> precedes the <ordinary-table-body>, none of the <ordinary-table-item-declarations> in the <ordinary-table-body> can contain a <table-preset>.

An <ordinary-entry-specifier> used in a <table-type-declaration> (see Section 2.2) must not contain a <table-preset>.

An <ordinary-table-body> must contain at least one <ordinary-table-item-declaration>.

A <packing-specifier> of N is permitted in a tightly-structured table only if the table entry contains only one item.

The number of words allocated for a table entry must not exceed MAXBITS/BITSINWORD.

2.1.2.4 SPECIFIED TABLE ENTRIES

Syntax:

```
<specified-entry-
    specifier>          ::= <words-per-entry>
                            <specified-item-description>
                            [<table-preset>]  ;              (2.1.6)

                          | <words-per-entry>
                            [<table-preset>]  ;              (2.1.6)
                            <specified-table-body>

<words-per-entry>       ::= W  [<entry-size>]

                          | V

<entry-size>            ::= <compile-time-integer-formula>   (5.1.1)

<specified-item-
    description>        ::= <item-type-description>  POS     (2.1.1)
                            (  <location-specifier>  )

<location-specifier>    ::= <starting-bit>  ,
                            <starting-word>

<starting-bit>          ::= <compile-time-integer-formula>   (5.1.1)

                          | *

<starting-word>         ::= <compile-time-integer-formula>   (5.1.1)

<specified-table-body>  ::= <specified-table-item-declaration>

                          | BEGIN
                            <specified-table-options>...
                            END

<specified-table-item-
    declaration>        ::= ITEM <table-item-name>           (2.1.2.3)
                            <specified-item-description>
                            [<table-preset>]  ;              (2.1.6)
```

<specified-table-options> ::= <specified-table-item-declaration>

| <null-declaration> (2.7)

Semantics:

A <specified-entry-specifier> is used to specify the contents of an entry of a specified table.

<Words-per-entry> specifies the size of (i.e, number of words in) each entry in the table. <Words-per-entry> containing a W indicates a fixed-length-entry specified table whereas V indicates a variable-length-entry specified table. In a fixed-length-entry specified table, <entry-size> (if present) specifies the number of words allocated to each entry in the table. In a tightly-structured table (in which <entry-size> must be omitted), the size of the entry is determined from the <structure-specifier>. In a variable-length-entry specified table, each entry is allocated one word.

The <location-specifier> specifies the physical location of the item from the start of the entry. <Starting-word> indicates at which word of the entry, starting from zero, the item is to start, and <starting-bit> indicates at which bit in the word, starting from zero at the leftmost part of the word, the item is to start. In the case of entries in tightly structured tables, <starting-bit> is considered to be relative to the start of the entry. A <starting-bit> of * indicates that the item should occupy the same amount of storage and be aligned in the same way it would if it were allocated outside a table, in order to ensure efficient access to the item. These rules apply to both fixed-length-entry and to variable-length-entry specified tables. Consequently, in a variable-length-entry table, reference to an item with subscript NN will reference that item relative to the start of the NNth entry, where each entry is considered to be one word long (i.e., a subscript of NN does not refer to the NNth logical entry in that table). It is entirely up to the programmer to keep track of the actual length of logical entries in such tables.

The <table-preset>, if present, specifies initial values for the table entries.

The value of unallocated bits in a <specified-table-entry> is implementation-dependent.

Constraints:

<Entry-size> must be omitted on tightly-structured tables and must be present otherwise.

<Entry-size> must be greater than zero and less than or equal to MAXBITS/BITSINWORD.

<Starting-word> must be non-negative. For items in tables with entry sizes specified by <entry-size>, <starting-word> plus number of words occupied by the

item must not exceed <entry-size>. For tightly structured tables <starting-word> must be zero.

<Starting-bit> must be non-negative and must not cause item position to violate other positioning constraints. For non-tightly structured tables it must also be less than BITSINWORD. For tightly structured tables <starting-bit> plus number of bits occupied by the item must not exceed <bits-per-entry>.

Only tables having STATIC allocation (implicitly or explicitly) may contain a <table-preset>.

Tables that are <formal-input-parameters> or <formal-output-parameters> (see Section 3.3) must not contain a <table-preset>.

If a <table-preset> precedes the <specified-table-body>, none of the <specified-table-item-declarations> in the <specified-table-body> can contain a <table-preset>. If any part of an item in a <specified-table-body> overlaps any part of another item in the table body, only one of the items can be preset.

A <specified-entry-specifier> used in a <table-type-declaration> (see Section 2.2) must not contain a <table-preset>.

A <specified-table-body> must contain at least one <specified-table-item-declaration>.

Non-character items whose size is one word or less cannot cross a word boundary. Character items, regardless of length, may start on any byte boundary, i.e., any value of the machine parameter BYTEPOS. Any <starting-bit> value is permitted for character items that do not cross word boundaries.

An implementation may restrict legal <starting-bit> values for pointer items that are initialized.

Variable-length-entry specified tables must contain a <specified-table-body>, and they cannot contain <table-presets> or <structure-specifiers>.

2.1.3 CONSTANT DECLARATIONS

Syntax:

```
<constant-declaration>    ::=  CONSTANT ITEM
                               <constant-item-name>
                               <item-type-description>      (2.1.1)
                               <item-preset>  ;             (2.1.6)

                          |    CONSTANT TABLE
                               <constant-table-name>
                               [<dimension-list>]           (2.1.2.1)
                               <table-description>          (2.1.2)
```

```
<constant-item-name>      ::= <name>                          (8.2.1)

<constant-table-name>     ::= <name>                          (8.2.1)
```

Semantics:

A <constant-declaration> creates an item or table whose value must be set by means of an <item-preset> or <table-preset> and whose value cannot be changed during execution of a program. The value of a constant item whose type class is not pointer can be used in a <compile-time-formula> (see Section 5.0). The value of a constant table or an item within a constant table may not be used in a <compile-time-formula>.

Physical storage will be allocated for all <constant-declarations> that are in <block-declarations>.

The allocation permanence of all allocated <constant-declarations> is considered to be STATIC, even if the declarations appear in a <subroutine-definition>.

2.1.4 BLOCK DECLARATIONS

Syntax:

```
<block-declaration>     ::= BLOCK <block-name>
                            [<allocation-specifier>] ;   (2.1.5)
                            <block-body-part>

                          | BLOCK <block-name>
                            [<allocation-specifier>]     (2.1.5)
                            <block-type-name>            (2.2)
                            [<block-preset>] ;           (2.1.6)

<block-name>            ::= <name>                       (8.2.1)

<block-body-part>       ::= <null-declaration>           (2.7)

                          | <data-declaration>           (2.1)

                          | BEGIN
                            <block-body-options>...
                            END

<block-body-options>    ::= <data-declaration>           (2.1)

                          | <overlay-declaration>        (2.6)


                          | <null-declaration>           (2.7)
```

Semantics:

A <block-declaration> declares a group of items, tables, and other blocks that are to be allocated in a contiguous area of storage. No allocation order is implied by the order of the declarations within a block unless the <block-body-part> contains an <order-directive>. If an <order-directive> is not in effect, <block-body-options> will be reordered, if necessary, to improve accessibility.

The <allocation-specifier> establishes the allocation permanence of blocks which are not enclosed in blocks. This allocation permanence is automatic if the declaration is in a subroutine and the <allocation-specifier> is omitted, otherwise it is STATIC (see Section 2.1.5). Blocks enclosed in blocks inherit the allocation permanence of the enclosing block.

The <block-declaration> describes the contents of the block either with a <block-type-name> (see Section 2.2) or with a <block-body-part>. The <block-body-part> contains explicit declarations of all the components of the block.

The <block-preset>, if present, specifies initial values for the block components. For <block-declarations> containing a <block-body-part> rather than a <block-type-name>, initial values may be specified with <block-presets>, <table-presets> and <item-presets> on the components themselves.

Constraints:

Only blocks having STATIC allocation (explicitly or by default) may contain a <block-preset> or a <data-declaration> containing an <item-preset>, <table-preset> or <block-preset>.

If a <constant-declaration> is in a block, the block must have STATIC allocation (explicitly or by default).

<Data-declarations> within a block must not contain an <allocation-specifier>.

Blocks that are <formal-input-parameters> or <formal-output-parameters> (see Section 3.3) must not contain an <allocation-specifier>, a <block-preset> or a <data-declaration> with an <item-preset>, <table-preset>, or <block-preset>.

Components of blocks declared with a <block-type-name> may be accessed only by using pointers.

2.1.5 ALLOCATION OF DATA OBJECTS

Syntax:

<allocation-specifier> ::= STATIC

Semantics:

Allocation of storage for a data object can be STATIC or automatic. STATIC allocation means that the data object is to exist throughout the entire execution of the program. Automatic allocation is applicable only to data declared within subroutines and means that the data object need only exist while the subroutine is executing (i.e., values are not necessarily preserved between calls).

STATIC is the default allocation for data not declared in subroutines and for all data declared in <external-declarations>. STATIC allocation can be explicitly specified both inside and outside of subroutines. Automatic is the default allocation for data declared in subroutines (other than <external-declarations>) and cannot be explicitly specified.

The treatment of STATIC data in a concurrent processing environment is implementation-dependent with respect to which data, if any, are shared among processes.

2.1.6 INITIALIZATION OF DATA OBJECTS

Syntax:

```
<item-preset>              ::=  =  <item-preset-value>

<item-preset-value>        ::=  <compile-time-formula>          (5.0)

                              |  <loc-function>                 (6.3.1)

<table-preset>             ::=  =  <table-preset-list>

<table-preset-list>        ::=  <default-preset-sublist>

                              |  [<default-preset-sublist>  ,]
                                 <specified-preset-sublist>,...

<default-preset-sublist>   ::=  <preset-values-option>,...

<specified-preset-
   sublist>                ::=  <preset-index-specifier>
                                 <preset-values-option>,...

<preset-index-specifier>   ::=  POS (  <constant-index>,...  )  :

<constant-index>           ::=  <compile-time-integer-formula>  (5.1.1)

                              |  <compile-time-status-formula>  (5.4)
```

```
<preset-values-option>      ::=  [<item-preset-value>]

                                 |  <repetition-count>
                                    (  <preset-values-option>,...  )

<repetition-count>          ::=  <compile-time-integer-formula>   (5.1.1)

<block-preset>              ::=  =  <block-preset-list>

<block-preset-list>         ::=  <block-preset-values-option>,...

<block-preset-values-
    option>                 ::=  <preset-values-option>

                                 |  [(  <table-preset-list>  )]

                                 |  [(  <block-preset-list>  )]
```

Semantics:

Items, tables, and blocks with STATIC allocation can be given initial values by means of <item-presets>, <table-presets>, and <block-presets>, respectively. Furthermore, constant items and tables must be given initial values with <item-presets> and <table-presets>. Initial values are values of the variables after a module has been loaded but prior to any dynamic reference to the variables. They do not imply any provision for later restoring values to the initial state.

An <item-preset> specifies an initial value for an item.

A <table-preset> specifies a list of initial values. If the <table-preset> occurs on an item within an entry of a table, the <table-preset> specifies values only for that item. If the table is dimensioned, the <table-preset> for the item, if present, may specify a list of values to initialize that item in each entry of the dimensioned table.

If the <table-preset> occurs on an entry of a table, the <table-preset> specifies values for all items within that entry. If the table is dimensioned, the <table-preset> specifies values for all the items in each entry of the dimensioned table. Assuming the entry has N items in it, the first N values in the <table-preset> are initial values for the N items in the first entry of the table (in the order in which the declarations appear), the second N values in the <table-preset> are initial values for the N items in the second entry of the table, etc.

Entries within a dimensioned table are normally initialized in order, the first entry being the one with the lowest value of each dimension index, and proceeding with the rightmost indices increasing most rapidly. This is the procedure followed when a <default-preset-sublist> is specified. If a <specified-preset-sublist> is used, initialization using the values in the sublist will start with the entry whose indices are specified in the <preset-index-specifier> and will proceed with the rightmost indices increasing most rapidly.

A <repetition-count> can be used as a shorthand to specify the number of consecutive repetitions of the sequence of <preset-values-options> enclosed in the parentheses following the <repetition-count>.

If a value is omitted in the <table-preset>, the item corresponding to the omitted value will remain uninitialized and cannot be given an initial value elsewhere in the preset.

A <block-preset> is used only to initialize a block declared with a <block-type-name>. The <block-preset-list> specifies initial values for the items, tables and blocks contained within the block in the order of their declaration. A parenthesized <table-preset-list> is used to initialize a contained table and a parenthesized <block-preset-list> is used to initialize a contained block. An omitted entry from the list indicates that the corresponding item, table, or block will remain uninitialized.

Constraints:

The type of each value in an <item-preset>, <table-preset> or <block-preset> must match or be implicitly convertible to the type of the data object being initialized (see Section 7.0).

The <preset-index-specifiers> within a <table-preset> must be specified such that no bit position is initialized more than once and the bounds of the table are not exceeded.

The value of the <repetition-count> must not be negative.

An item must not be initialized more than once by initializing another item that overlaps it.

The type of each <constant-index> in a <preset-index-specifier> must be equivalent to, or be implicitly convertible to, the type of the bounds of the corresponding dimension in the <dimension-list> of the declaration of the table.

The number of <constant-indices> in a <preset-index-specifier> must be the same as the number of <dimensions> in the table's <dimension-list>.

The allocation permanence of the argument of a <loc-function> used as a preset value must not be automatic; any index, if present, must be a <constant-index>.

2.2 TYPE DECLARATIONS

Syntax:

```
<type-declaration>        ::= <item-type-declaration>
                            | <table-type-declaration>
                            | <block-type-declaration>
```

```
<item-type-declaration>    ::=  TYPE  <item-type-name>
                                <item-type-description>  ;     (2.1.1)

<item-type-name>           ::=  <name>                          (8.2.1)

<table-type-declaration>   ::=  TYPE  <table-type-name>
                                TABLE  <table-type-specifier>

<table-type-specifier>     ::=  [<dimension-list>]              (2.1.2.1)
                                [<structure-specifier>]         (2.1.2.2)
                                [<like-option>]
                                <entry-specifier>               (2.1.2)

                             |  [<dimension-list>]              (2.1.2.1)
                                <table-type-name>  ;

<table-type-name>          ::=  <name>                          (8.2.1)

<like-option>              ::=  LIKE  <table-type-name>

<block-type-declaration>   ::=  TYPE  <block-type-name>
                                BLOCK  <block-body-part>        (2.1.4)

<block-type-name>          ::=  <name>                          (8.2.1)
```

Semantics:

A <type-declaration> is used to describe the attributes of a data object, except for allocation, and associate the <type-name> with the attributes. Later <data-declarations> may allocate space for one or more objects with those attributes.

An <item-type-declaration> associates the <item-type-name> with the <item-type-description>.

A <table-type-declaration> associates the <table-type-name> with the <table-type-specifier>.

A table type is dimensioned if its <table-type-specifier> contains a <dimension-list> or references a dimensioned table type (either via a <like-option> or in lieu of containing an <entry-specifier>).

If a <like-option> is specified, the entry being described consists of the items in the type named in the <like-option> together with the items in the <entry-specifier>. The physical positioning of items in the <like-option> relative to the start of the entry is fixed at the time the <like-option> type name is declared and is not changed by its use as a <like-option>. If the table is an ordinary table, the <packing-specifier>, if present, only applies to the items in the <entry-specifier>, not to the items obtained from the <like-option>. If the table is a specified table, the <words-per-entry> in the <entry-specifier>, if present, specifies the total size of the entry including

the items obtained from the <like-option>. If the <table-type-declaration> contains a <structure-specifier> of T, <bits-per-entry> specifies the total number of bits the entry is to occupy including items obtained from the <like-option>. If <bits-per-entry> is omitted it will default to the minimum number of bits needed to store the entry, including items obtained from the <like-option>.

The physical representation of a table type is fixed by the <table-type-declaration>. All objects allocated with such a type name will have the same representation. In particular, the position of table items in a <like-option> is not modified by the occurrence of a <packing-specifier> or <order-directive> in the <entry-specifier>. However, unused space in the portion described by the <like-option> can be occupied by table items given in a packed <entry-specifier>.

A <block-type-declaration> associates the <block-type-name> with the <block-body-part>.

For type matching purposes, a type name is considered to be an abbreviation for its associated <item-type-description>, <table-type-specifier>, or <block-body-part>, in any context except within a <pointer-item-description>.

Constraints:

The <item-type-description>, <table-type-specifier>, or <block-body-part> in a <type-declaration> must not contain an <item-preset>, <table-preset>, or <block-preset>.

A <block-body-part> in a <block-type-declaration> cannot contain <constant-declarations>.

A <table-type-specifier> must not both contain a <dimension-list> and reference a dimensioned table type (either via a <like-option> or in lieu of containing an <entry-specifier>).

Tables may be characterized as parallel, serial, tight, ordinary, variable-length-entry, and specified. The characterizations of the table type in a <like-option> must be the same as those of the <table-type-declaration> in which the <like-option> appears.

<Words-per-entry> of the <table-type-specifier> must not specify a value that is less than <words-per-entry> of the type name specified in a <like-option>.

The (explicit or default) number of bits in an entry in a <table-type-specifier> having tight structure must not be less than the number of bits in an entry of the <table-type-name> specified in a <like-option>.

A <table-type-name> must be a <name> declared in a <table-type-declaration>.

A <block-type-name> must be a <name> declared in a <block-type-declaration>.

Note:

A <table-type-name>, <item-type-name>, or <block-type-name> must not be a formal parameter name or an actual parameter name.

2.3 STATEMENT NAME DECLARATIONS

Syntax:

```
<statement-name-declaration>    ::= LABEL
                                    <statement-name>,... ;     (4.0)
```

Semantics:

A <statement-name-declaration> is used to explicitly declare a <statement-name>. Ordinarily, a <statement-name> is implicitly declared by its use in a <label>. An explicit <statement-name-declaration>, however, must be used for statement name <formal-input-parameters>, for statement names that are the same as <define-names> declared in an enclosing scope, and for external <statement-name-declarations>.

Constraints:

Each <statement-name> in a <statement-name-declaration> must either be a <formal-input-parameter> to the subroutine containing the <statement-name-declaration> or else must be used in a <label> in the immediate scope containing the <statement-name-declaration> (i.e., not including nested scopes), or else the <statement-name-declaration> must be a <ref-specification-choice>.

2.4 DEFINE DECLARATIONS

Syntax:

```
<define-declaration>          ::= DEFINE <define-name>
                                  <definition-part>

<define-name>                 ::= <name>                              (8.2.1)

<definition-part>             ::= [<formal-define-parameter-list>]
                                  <define-string> ;

<formal-define-parameter-
  list>                       ::= ( <formal-define-parameter>,... )

<formal-define-parameter>     ::= <letter>                            (8.1)

<define-string>               ::= " [<character>...] "                (8.1)
```

Semantics:

A <define-declaration> is used to associate a name with a (possibly parameterized) text string, the <define-string>. The <define-string> will be substituted for the <define-name> when the <define-name> is used in a <define-call> (see Section 2.4.1).

The <formal-define-parameter-list> is used to declare <formal-define-parameters>. These parameters receive values from <actual-define-parameters> in each <define-call> (see Section 2.4.1). The values are substituted in the <define-string> wherever the <formal-define-parameters> are referenced. Reference to a <formal-define-parameter> within the <define-string> is indicated by preceding the parameter name with an exclamation point. Such parameter references can occur anywhere within the <define-string> and, by appropriate juxtaposition, can be used to create new symbols.

Within the <define-string>, the quotation mark (") and exclamation point (!) can be used as simple characters by doubling them. A <define-string> is terminated by the first undoubled quotation mark, regardless of the lexical context in which the undoubled quotation mark appears.

As with other <names>, a <define-name> is known in the scope containing its declaration and may be redeclared in an inner scope.

The <define-string> may contain <define-calls>. Such calls will be expanded for each substitution of the <define-string>, using the definition active in the scope of the <define-call>.

Constraints:

A <comment> delimited by quotation marks must not occur in a <define-declaration> between the <define-name> and the <define-string>.

Circular <define-declarations> as the result of <define-strings> containing <define-calls> are not allowed.

The same <letter> must not appear more than once in any <formal-define-parameter-list>.

2.4.1 DEFINE CALLS

Syntax:

<define-call> ::= <define-name> [<actual-define-parameter-list>] (2.4)

<actual-define-parameter-list> ::= (<actual-define-parameter>,...)

<actual-define-parameter> ::= [<character>...] (8.1)

| " [<character>...] " (8.1)

Semantics:

A <define-call> is used to cause textual substitution to occur. A <define-call> is processed as follows:

1. The characters comprising <actual-define-parameters> are substituted for the corresponding <formal-define-parameters> in the <define-string> associated with the <define-name>.

2. The resulting <define-string> is scanned from its beginning to determine what <symbols> it contains; these <symbols> are processed as though they had appeared in the original text at the point of the replaced <define-call>.

3. Scanning of the source program continues as if the <symbols> found in the <define-string> had occurred in the original text.

Note that the substituted source text may be found to contain <define-calls> and these are processed in the same manner.

If an <actual-define-parameter> is omitted, a null string will be substituted for the <formal-define-parameter>. If the number of <formal-define-parameters> exceeds the number of <actual-define-parameters>, null strings will be substituted for the trailing <formal-define-parameters>.

If the first non-blank character of the <actual-define-parameter> is not a quotation mark, the parameter is the entire string of characters starting with this character and ending at, but not including, either (1) the first right parenthesis that is not balanced by a left parenthesis that is part of the <actual-define-parameter>, or (2) the first comma that is not between such balanced parentheses. All characters within the parameter, including quotation marks, represent themselves.

If the first non-blank character of the <actual-define-parameter> is a quotation mark, the parameter consists of all characters enclosed by this quotation mark and the next succeeding undoubled quotation mark. Characters within the parameter represent themselves except doubled quotation marks ("") which represent a single quotation mark (").

<Define-calls> are not recognized in <comments> and <character-literals>.

Constraints:

The <actual-define-parameter-list> must not be omitted if the corresponding <define-declaration> contains a <formal-define-parameter-list>.

The number of <actual-define-parameters> must not be greater than the corresponding number of <formal-define-parameters>.

A <define-call> cannot be juxtaposed with surrounding symbols so as to create new symbols after substitution. In particular, the resulting <define-string> produced by a <define-call> must not end with an incomplete bracketed symbol (<character-literal>, <bit-literal>, <comment>, <define-string>, or quoted <actual-define-parameter>).

A <define-call> must not be used as the <name> being declared within a declaration.

A <define-call> must not be used as a <formal-input-parameter> or <formal-output-parameter> within a <procedure-heading> or <function-heading>.

Note:

The define-listing directives (see Section 9.7.2) allow programmer control over whether the source program listing contains the expanded string, the define invocation, or both, for <define-calls>.

Examples:

```
DEFINE FOO(A) "BAZ !AFAZ !A";
DEFINE BAR "HELLO";
DEFINE BARFAZ "GOODBYE";
DEFINE HELLOFAZ "NOT USED";
FOO(BAR)
```

The result of the <define-call>, FOO(BAR), after substituting for the formal parameter !A is BAZ BARFAZ BAR, and after rescanning this string, the final result is BAZ GOODBYE HELLO.

```
DEFINE MAKEDEF(N, S) "DEFINE !N ""!S""";
MAKEDEF(NEW, HELLO);
```

The result is DEFINE NEW "HELLO"; , i.e., a new <define-declaration>.

2.5 EXTERNAL DECLARATIONS

Syntax:

```
<external-declaration>     ::=  <def-specification>          (2.5.1)
                              |  <ref-specification>          (2.5.2)
```

Semantics:

<External-declarations> declare <names> that are potentially known in other <modules> of the <complete-program>. Such names are said to be external.

Constraint:

Formal parameter names cannot be declared external.

2.5.1 DEF SPECIFICATIONS

Syntax:

```
<def-specification>          ::=  <simple-def>
                              |  <compound-def>

<simple-def>                 ::=  DEF
                                  <def-specification-choice>

<compound-def>               ::=  DEF  BEGIN
                                  <def-specification-choice>...
                                  END

<def-specification-choice>   ::=  <null-declaration>            (2.7)
                              |  <data-declaration>            (2.1)
                              |  <def-block-instantiation>
                              |  <statement-name-declaration>  (2.3)


<def-block-instantiation>    ::=  BLOCK  INSTANCE
                                  <block-name>  ;              (2.1.4)
```

Semantics:

<Def-specifications> enable data objects to be declared that are potentially available via <ref-specifications> and/or <compool-directives> for use in other <modules>. Physical storage will be allocated for these objects.

Either a <def-block-instantiation> or a <block-declaration> may be used in a <def-specification> to create a block with external scope. However, in order for a <def-block-instantiation> to be meaningful, a <ref-specification> containing a <block-declaration> having the same <block-name> must exist, either in that <module> or in a <compool-module> that is referenced via a <compool-directive>. Preset information used in the creation of a block declared with a <def-block-instantiation> will be obtained from the corresponding <ref-specification>.

A <statement-name-declaration> in a <def-specification> makes the addresses of the designated statements available as linkage information in the environment of the <complete-program> but does not make these names available as targets of out-of-scope GOTO statements (see Section 4.7).

<Data-declarations> in <def-specification-choices> have STATIC allocation, whether or not explicitly specified.

Constraints:

A <data-declaration> in a <def-specification> and a corresponding declaration in a <ref-specification> must agree in name, type, and all attributes. However, a compiler will perform this check across <module> boundaries only if a connection is established between the <modules> via a <compool-directive>.

The <data-declaration> in a <def-specification> cannot be a <constant-declaration>. (This constraint does not prevent <constant-declarations> from appearing in <block-declarations> in <def-specifications>.)

2.5.2 REF SPECIFICATIONS

Syntax:

```
<ref-specification>     ::= <simple-ref>
                          | <compound-ref>

<simple-ref>            ::= REF
                            <ref-specification-choice>

<compound-ref>          ::= REF  BEGIN
                            <ref-specification-choice>...
                            END
```

```
<ref-specification-choice>  ::=  <null-declaration>              (2.7)
                             |   <data-declaration>              (2.1)
                             |   <subroutine-declaration>        (3.0)
                             |   <statement-name-declaration>    (2.3)
```

Semantics:

A <ref-specification> enables a <module> to reference a <name> whose <def-specification> is in another <module>.

Physical storage for external names occurs in the <module> containing the <def-specification> and not in the <module> containing the <ref-specification>.

A <ref-specification> for a <name> may appear in a <compool-module> and the corresponding DEF in another <module>. In that case, the <name> will be available for use in any other module of the <complete-program>, provided that the referencing module has the appropriate <compool-directive>. The compiler will enforce the requirement that the DEF and the REF specifications agree, provided the <module> containing the DEF has the appropriate <compool-directive>. Alternatively, the <ref-specification> may appear in the accessing <module> directly instead of in a compool (bypassing the compool entirely), but in this case it will be beyond the compiler's ability to check for compatibility between the DEF and the REF specifications. When <ref-specifications> are used outside of compools to gain access to external names, the programmer is entirely responsible for the correct usage of those names.

For <data-declarations> in a <def-specification> that is in a <compool-module>, no <ref-specification> is necessary if the accessing module has a <compool-directive> that causes that data to be imported (see Section 9.1).

<Data-declarations> in <ref-specification-choices> have STATIC allocation, whether or not explicitly specified.

Constraints:

For every <data-declaration> in a <ref-specification>, there must exist a corresponding declaration in a <def-specification>. For every <subroutine-declaration> in a <ref-specification>, there must exist in some <procedure-module> or <main-program-module> a corresponding <subroutine-definition> preceded by DEF.

In a <ref-specification>, presets are illegal in <item-declarations> and <table-declarations> and are optional in <block-declarations>. A <ref-specification> that contains presets can be used only in conjunction with a <def-block-instantiation>.

A <data-declaration> in a <ref-specification> cannot be a <constant-declaration>. (This constraint does not prevent <constant-declarations> from appearing in <block-declarations> in <ref-specifications>).

2.6 OVERLAY DECLARATIONS

Syntax:

```
<overlay-declaration>   ::= OVERLAY
                            [<absolute-address>]
                            <overlay-expression> ;

<absolute-address>      ::= POS ( <overlay-address> ) :

<overlay-address>       ::= <compile-time-integer-formula>    (5.1.1)

<overlay-expression>    ::= <overlay-string>:...

<overlay-string>        ::= <overlay-element>,...

<overlay-element>       ::= <spacer>
                          | <data-name>
                          | ( <overlay-expression> )

<spacer>                ::= W <compile-time-integer-formula> (5.1.1)

<data-name>             ::= <item-name>                      (2.1.1)
                          | <table-name>                     (2.1.2)
                          | <block-name>                     (2.1.4)
```

Semantics:

An <overlay-declaration> is used to specify any or all of the following:

1) that data objects are to have a specific allocation order

2) that certain data objects are to occupy the same memory locations as other data objects

3) that certain data objects are to be allocated at a particular absolute memory location

The <overlay-elements> in an <overlay-string> are allocated memory locations in the order of their appearance in the string. The memory locations allocated to the elements of an <overlay-string> that appears to the left of a colon in an <overlay-expression> are overlayed with the space allocated to the elements of the <overlay-string> that appears to the right of the colon.

The <overlay-address> specifies an absolute memory location at which allocation of the <overlay-expression> begins. The meaning of an <overlay-address> is implementation-dependent and machine dependent.

A <spacer> in an <overlay-string> specifies a number of words to be skipped during allocation.

Constraints:

The allocation permanence of all data objects in an <overlay-declaration> must be the same (i.e., all STATIC or all automatic).

An <overlay-declaration> within a <block-declaration> or <block-type-declaration> must not reference data names declared outside the block or block type or within nested blocks.

An <overlay-declaration> outside a <block-declaration> or <block-type-declaration> must not reference data names declared within a block or block type.

An <overlay-declaration> within a <block-declaration> or <block-type-declaration> must not include an <absolute-address>.

A <block-declaration> or <block-type-declaration> must not include an <overlay-declaration> if an <order-directive> is in effect for the block or block type.

Declarations for all <data-names> in an <overlay-declaration> must precede the <overlay-declaration>, and all must be in the same scope.

<Overlay-declarations> cannot be used to specify more than one physical location to any data object.

Names of formal parameters cannot be used in <overlay-declarations>.

If an <overlay-address> is specified, all <data-names> used in the <overlay-expression> must have an (explicit or default) allocation permanence of STATIC.

Note:

A <data-name> in an <overlay-declaration> cannot be declared in a <constant-declaration>.

2.7 NULL DECLARATIONS

Syntax:

```
<null-declaration>          ::=  ;
                                 |  BEGIN  END
```

Semantics:

A <null-declaration> has no semantic effect.

3.0 PROCEDURES AND FUNCTIONS

Syntax:

<subroutine-declaration>	::=	<procedure-declaration>	(3.1)
	|	<function-declaration>	(3.2)
<subroutine-definition>	::=	<procedure-definition>	(3.1)
	|	<function-definition>	(3.2)

Semantics:

Subroutines describe algorithms that may be executed from more than one place in a <complete-program>. A subroutine is either a procedure, which is invoked by a <procedure-call-statement>, or a function, which is invoked by a <function-call>.

A <subroutine-definition> contains the executable code for the subroutine, in addition to declarations for all local data and formal parameters, as well as definitions of any nested subroutines. A <subroutine-definition> is said to define the subroutine.

A <subroutine-declaration>, on the other hand, is said to declare the subroutine. A <subroutine-declaration> contains the heading of the subroutine and <declarations> for the formal parameters, but it contains no executable code. A <subroutine-declaration> is required in a <ref-specification> for each subroutine that is invoked in a module other than the module containing its definition (see Section 2.5). A <subroutine-declaration> is also required in two other situations: (1) when a subroutine name is declared as a formal parameter (see Section 3.3), and (2) when the name of the subroutine is the same as a <define-name> in an enclosing scope. It is not necessary to provide a <subroutine-declaration> if the subroutine is invoked only in the <module> where it is defined and if its name is not passed as a parameter or used in an enclosing scope as a <define-name>.

Constraints:

The <procedure-heading> or <function-heading> of a <subroutine-declaration> and that of the corresponding <subroutine-definition> must have identical attributes, and the parameters (both input and output) must agree in number, type, and order. (This constraint will be enforced only when the declaration is known in the scope of the definition.) Also, the subroutine name in the declaration and definition must be the same (unless the <subroutine-declaration> is for a formal parameter).

3.1 PROCEDURES

Syntax:

```
<procedure-declaration>     ::=  <procedure-heading>  ;
                                 <declaration>                    (2.0)

<procedure-definition>      ::=  <procedure-heading>  ;
                                 <procedure-body>

<procedure-heading>         ::=  PROC
                                 <procedure-name>
                                 [<subroutine-attribute>]
                                 [<formal-parameter-list>]        (3.3)

<subroutine-attribute>      ::=  REC

                               |  RENT

<procedure-name>            ::=  <name>                           (8.2.1)

<procedure-body>            ::=  <subroutine-body>

<subroutine-body>           ::=  <statement>                      (4.0)

                               |  BEGIN [<declaration>...]        (2.0)
                                  <statement>...                 (4.0)
                                  [<subroutine-definition>...]   (3.0)
                                  [<label>...] END               (4.0)
```

Semantics:

The <procedure-heading> in a <procedure-declaration> may contain a <formal-parameter-list>, which specifies the names that are used in the <procedure-body> to refer to the corresponding arguments supplied by each call of the procedure. The syntax, semantics, and constraints for a procedure's <formal-parameter-list> are the same as for a function's <formal-parameter-list>, and are presented in Section 3.3.

The differences between a <procedure-declaration> and a <procedure-definition> are described in Section 3.0.

A <subroutine-attribute> of REC indicates that the subroutine is potentially recursive, i.e., that at run time, an invocation of the subroutine may be dynamically nested within another invocation of it. If REC is present, physical allocation of locally-declared automatic data will occur dynamically. The data will be allocated and deallocated when the subroutine is entered and exited, respectively. This assures that separate copies of the local data will exist for each successive call in the recursive chain. Locally-declared STATIC data, however, will be allocated once, and the same storage will be used for all calls of that subroutine throughout the <complete-program>.

A <subroutine-attribute> of RENT indicates that the subroutine is re-entrant and may therefore be executed concurrently in a concurrent processing environment. A recursive subroutine is also re-entrant.

If execution of the <procedure-body> is completed without executing a RETURN statement, an ABORT statement, or a GOTO statement whose target is the name of a formal parameter, an implicit RETURN statement is executed.

Constraints:

A <procedure-declaration> can contain no <declarations> other than those for the procedure's formal parameters. <Declarations> of local data appear only in the procedure's definition.

A procedure must not be invoked recursively if it is not declared REC.

A procedure must not be invoked re-entrantly if it is not declared RENT or REC.

A <subroutine-body> must contain at least one non-null <statement> (e.g., RETURN).

3.2 FUNCTIONS

Syntax:

<function-declaration>	::=	<function-heading> ; <declaration>	(2.0)
<function-definition>	::=	<function-heading> ; <function-body>	
<function-heading>	::=	PROC <function-name> [<subroutine-attribute>] [<formal-parameter-list>] <item-type-description>	(3.1) (3.3) (2.1.1)
<function-name>	::=	<name>	(8.2.1)
<function-body>	::=	<subroutine-body>	(3.1)

Semantics:

The differences between a <function-declaration> and a <function-definition> are described in Section 3.0.

The <item-type-description> specifies the type of the return value of the function. Within the body of the function, the name of the function may be assigned to as a variable in an assignment statement. When the function is exited, the most recent value assigned to the <function-name> is used as the

value of the function. The return value is considered to be allocated as automatic storage (see Section 2.1.5).

Use of the <function-name> in a <formula> within the body of the function is a recursive invocation of the function. Within the body of the function, the <function-name> may also be used as an <actual-input-parameter> in a subroutine call when the corresponding <formal-input-parameter> is a <function-name> (see Section 3.3).

The <function-heading> in a <function-declaration> or <function-definition> may contain a <formal-parameter-list>, which specifies the names that are used in the <function-body> to refer to the corresponding arguments supplied by each call of the function. The syntax, semantics, and constraints for a function's <formal-parameter-list> are the same as for a procedure's <formal-parameter-list>, and are presented in Section 3.3.

The inclusion of an <item-type-description> in the heading of a subroutine indicates that the subroutine is a function.

The <subroutine-attributes> of REC and RENT apply to functions in the same way as for procedures (see Section 3.1).

If execution of the <function-body> is completed without executing a RETURN statement, an ABORT statement, or a GOTO statement whose target is the name of a formal parameter, an implicit RETURN statement is executed.

Constraints:

The <function-name> may not be used as an <actual-output-parameter>.

The <function-name> is not declarable as a <name> within the function body.

A <function-declaration> can contain no <declarations> other than those for the function's formal parameters. <Declarations> of local data appear only in the function's definition.

A function must not be invoked recursively if it is not declared REC.

A function must not be invoked re-entrantly if it is not declared RENT or REC.

The <function-name> must be assigned a value before the function is exited via an explicit or implicit RETURN statement.

3.3 PARAMETERS OF PROCEDURES AND FUNCTIONS

Syntax:

```
<formal-parameter-list>  ::=  (  [<formal-input-parameter>,...]
                                 [  :  <formal-output-parameter>,...  ] )
```

```
<formal-input-parameter>  ::=  [<parameter-binding>]
                               <input-parameter-name>

<formal-output-parameter> ::=  [<parameter-binding>]
                               <output-parameter-name>

<parameter-binding>       ::=  BYVAL

                               |  BYREF

                               |  BYRES

<input-parameter-name>    ::=  <data-name>                    (2.6)

                               |  <statement-name>             (4.0)

                               |  <subroutine-name>

<output-parameter-name>   ::=  <data-name>                    (2.6)

<subroutine-name>         ::=  <procedure-name>

                               |  <function-name>              (3.2)
```

Semantics:

Parameters permit subroutines to have locally-declared <names> that correspond to entities whose values can be different for different calls.

<Formal-input-parameters> and <formal-output-parameters> constitute the formal parameters of the subroutine. When the subroutine is invoked, the formal parameters are associated with a corresponding list of actual parameters supplied in the subroutine call (see Section 4.5).

<Formal-input-parameters> transfer values into the <subroutine-body> from the corresponding <actual-input-parameters>. <Formal-output-parameters> transfer values into the <subroutine-body> and also transfer values from the <subroutine-body> back to the corresponding <actual-output-parameters>.

If a formal parameter is a <data-name> it may be bound to the corresponding actual parameter in any of the following ways: by reference, by value, by result, or by value-result.

Reference binding means that the actual parameter and the formal parameter denote the same physical object. Any change in the value of the formal parameter entails an immediate change in the value of the actual parameter and vice-versa.

Value-result binding means that the formal parameter denotes a separate data object, assigned the value of the actual parameter on entry to the subroutine, and used to assign its value to the actual parameter on normal exit from the

subroutine. Since it is a separate data object there is no interaction between it and the actual parameter during execution of the subroutine.

Value binding is similar except the actual parameter is not modified on exit from the subroutine.

Result binding leaves the value of the formal parameter undefined on entry to the subroutine but is otherwise like value-result binding.

Standard rules for types of binding indicate the effect normally required:

Reference binding shall be used for blocks, tables, and for entries of all except tight tables.

Value binding shall be used for input items and tight table entries.

Value-result binding shall be used for output items and tight table entries.

Explicit <parameter-binding> specification affects these rules as follows:

BYREF - reference binding is required. If the actual parameter cannot be passed by reference (such as a badly aligned table item), the compiler shall allocate a temporary variable, use value or value-result binding as appropriate to pass the parameter between its actual location and the temporary variable, and pass the temporary variable by reference to the subroutine.

BYVAL - reference binding is prohibited. The parameter shall be passed by value or value-result as appropriate.

BYRES - result binding is required.

An implementation may optimize binding methods provided it guarantees required results both in parameters passed and in side effects, if any.

If the <formal-input-parameter> is a <statement-name>, the semantics are as described in Section 4.7.

If the <formal-input-parameter> is a <subroutine-name> the <name> of the corresponding actual parameter determines which <subroutine-definition> to associate with the formal parameter's <subroutine-declaration> on each call. A call to that subroutine via the formal parameter <name> will be treated as if the corresponding actual parameter subroutine had been called from the same environment in which <subroutine-name> was originally specified as an <actual-input-parameter>.

The order of evaluation of actual parameters is unspecified.

In the absence of an <interference-directive>, no interference is assumed within the subroutine between actual parameter data and formal table or block parameters, or between actual parameters and variables accessed directly from within the subroutine.

Constraints:

The same name must not appear more than once in any <formal-parameter-list>.

A <formal-input-parameter> cannot be used in a context in which its value can be altered (e.g., as a target in an <assignment-statement>).

Names of data declared as formal parameters must not be used in <overlay-declarations>.

Declarations of formal parameters must not contain <allocation-specifiers> or presets.

<External-declarations> of formal parameters are not permitted.

The <subroutine-definition> (and <subroutine-declaration>, if one is present) must contain an explicit <declaration> for each <name> in the <formal-parameter-list>.

For any subroutine call, the number of formal and actual input parameters must be the same, and the number of formal and actual output parameters must be the same.

Declarations of formal parameters cannot be <constant-declarations> or <type-declarations>.

For all table parameters, the types of the formal parameters and those of the corresponding actual parameters must be equivalent (see Section 7.0). This requirement extends to the types and associated attributes of all components, and their allocation order. For all item parameters, the rules for implicit attribute conversion apply (see Section 7.0). Block parameters match under the following conditions: (1) the components must have equivalent types and must be in the same textual order; (2) an !ORDER directive is either present in both <block-body-parts> or absent in both <block-body-parts>; and (3) <overlay-declarations> in both blocks have the same effect.

The actual parameter corresponding to a formal input parameter <statement-name> must be a <statement-name> and must conform to the constraints specified in Section 4.7 for <statement-names> used in <goto-statements>.

The actual parameter corresponding to a formal input parameter <subroutine-name> must be the name of a subroutine. Parameter types and return value types of formal and actual subroutines must match exactly, as described in Section 3.0 constraints.

BYRES binding must not be specified for input parameters.

Notes:

Throughout this specification, 'formal parameter' refers to entities named in a <formal-parameter-list>, but not to their components. For example, an item declared in a formal parameter block is not a formal parameter.

3.4 INLINE PROCEDURES AND FUNCTIONS

Syntax:

<inline-declaration> ::= INLINE
<subroutine-name>,... ; (3.1)

Semantics:

An <inline-declaration> causes the object code for the bodies of each of the designated subroutines to be inserted at the point of every call of that subroutine within the scope containing the <inline-declaration>. This will be done instead of inserting code for calling a remote subroutine body.

The effect of the <inline-declaration> extends for just the name scope in which the <inline-declaration> appears. It does not affect calls appearing in enclosing scopes.

If any actual parameters to inline subroutines are constants, inline expansion may cause some formulas in the <subroutine-bodies> to become evaluable at compile time. Compile-time evaluation of these formulas will be performed and any corresponding error messages will be generated as though the programmer had written those formulas directly. Except for the effects of compile-time evaluation, the semantics of inline subroutine expansion are identical to the semantics of the normal, remote subroutine call mechanism.

Inline subroutines may themselves contain (possibly inline) subroutine calls, but they may not contain nested subroutine definitions.

Inline subroutine names may be used as actual parameters, but a call to the matching formal parameter name will result in a closed rather than inline invocation (even if the actual parameter is an inline subroutine).

Constraints:

Names of subroutines whose definitions appear in another module cannot be used in <inline-declarations>.

Formal parameters cannot be declared to be inline.

It is illegal to have an inline subroutine invocation of a subroutine that is already being expanded inline.

Formal parameters of inline subroutines cannot be used in contexts where the syntax requires a compile-time formula.

3.5 MACHINE-SPECIFIC PROCEDURES AND FUNCTIONS

Semantics:

Each compiler implementation may provide a set of procedures and functions that are intrinsically recognized by the compiler. These procedures and functions shall typically encompass operations that are not directly provided by the language. They may be implemented as subroutines or via inline code, whichever is suitable. The use of inline code is particularly suitable as a vehicle for invoking single machine instructions which are peculiar to the target machine.

In general, a subroutine will be provided for machine instructions whose execution would otherwise be unobtainable through the language. It is not intended that every target machine instruction be supported as a machine-specific procedure or function. Subroutines will, however, be provided for machine-specific instructions whose meaning is not expressible in the language (e.g., "load status word", "test condition code"), as well as instructions for which a J73 subroutine could be written but which are directly implemented by target-machine instructions (e.g., "sine", "matrix multiply", or "rotate length-32 bitstring", etc.). Such subroutines will be defined at system scope and hence their names will be redefinable in inner scopes. Such subroutines will be invoked in the same way as other subroutines (see Sections 4.5 and 6.3). The particular parameters to such subroutines are subroutine-dependent.

Implementation requirements for each such subroutine include specification of the operation to be performed and of the rules for each formal parameter, including both its JOVIAL attributes and how it is used. The compiler shall generate code to use the parameters and perform the specified operation.

4.0 STATEMENTS

Syntax:

```
<statement>              ::= [<label>...]
                             <simple-statement>

                           | [<label>...]
                             <compound-statement>

<simple-statement>       ::= <assignment-statement>          (4.1)

                           | <loop-statement>                (4.2)

                           | <if-statement>                  (4.3)

                           | <case-statement>                (4.4)

                           | <procedure-call-statement>      (4.5)

                           | <return-statement>              (4.6)

                           | <goto-statement>                (4.7)

                           | <exit-statement>                (4.8)

                           | <stop-statement>                (4.9)

                           | <abort-statement>               (4.10)

                           | <null-statement>

<null-statement>         ::= ;

                           | BEGIN [<label>...]
                             END

<label>                  ::= <statement-name> :

<statement-name>         ::= <name>                          (8.2.1)

<compound-statement>     ::= BEGIN <statement>...
                             [<label>...] END
```

Semantics:

<Statements> are the means by which computational algorithms are specified. They control the execution of the <complete-program>.

A <compound-statement> permits a sequence of <statements> to be used in contexts requiring a single <statement>.

A <null-statement> results in no operation.

A <label> is used to attach a <statement-name> to a <statement>. A <label> that is attached to the END of a <compound-statement> or <null-statement> is treated as if a no operation <statement> followed the <label>.

4.1 ASSIGNMENT STATEMENTS

Syntax:

<assignment-statement>	::=	<variable-list> = <formula> ;	(5.0)
<variable-list>	::=	<variable>,...	(6.1)

Semantics:

An <assignment-statement> causes the value of the <formula> to the right of the equal sign to be assigned to the <variables> to the left of the equal sign.

In performing the assignment, the <formula> is evaluated first. Then, the leftmost variable is evaluated and the value of the formula is assigned to that variable. Next, the second-to-the-left variable is evaluated and the value of the formula is assigned to it. This sequence of evaluations continues until the list of variables is exhausted. If necessary and permitted (see Section 7.0), the value of the formula is implicitly converted to the type of the variable being assigned to. For numeric values, the value is rounded or truncated according to the <round-or-truncate> attribute of each variable being assigned to (see Sections 2.1.1.2 and 2.1.1.3).

Constraints:

The type of the <formula> must match or be implicitly convertible to that of each of the <variables> according to the rules given in Section 7.0.

All <variables> in the <variable-list> must be of the same type class.

None of the <variables> may be <formal-input-parameters>.

Note:

Assignment semantics and constraints apply to presets (Section 2.1.6), assignments to <control-items> in <loop-statements> (Section 4.2), and some types of actual/formal parameter correspondence (Section 3.3).

Since the implemented precision of packed fixed point table items may be less than the implemented precision of an unpacked item having the same fixed type, and since rounding and truncation are performed with respect to implemented precision, assignment to packed table items may change the value being assigned (see Section 7.0).

4.2 LOOP STATEMENTS

Syntax:

```
<loop-statement>          ::= <loop-type>
                              <controlled-statement>

<loop-type>               ::= <while-clause>

                            | <for-clause>

<controlled-statement>    ::= <statement>

<while-clause>            ::= WHILE <boolean-formula> ;        (5.2.2)

<for-clause>              ::= FOR <control-item> :
                              <control-clause> ;

<control-item>            ::= <control-variable>

                            | <control-letter>

<control-variable>        ::= <item-name>                      (2.1.1)

<control-letter>          ::= <letter>                         (8.1)

<control-clause>          ::= <initial-value>
                              [<continuation>]

<initial-value>           ::= <formula>                        (5.0)

<continuation>            ::= <by-or-then phrase>
                              [<while-phrase>]

                            | <while-phrase>
                              [<by-or-then-phrase>]

<by-or-then-phrase>       ::= <by-phrase>

                            | <then-phrase>

<by-phrase>               ::= BY <by-formula>

<by-formula>              ::= <numeric-formula>                (5.1)

<then-phrase>             ::= THEN <formula>                   (5.0)

<while-phrase>            ::= WHILE <boolean-formula>          (5.2.2)
```

Semantics:

A <loop-statement> provides for the iterative execution of a statement.

If the <while-clause> form of the <loop-statement> is used, the <controlled-statement> is executed until the value of the <boolean-formula> becomes FALSE. The <boolean-formula> is evaluated before each iteration.

If the <for-clause> form is used, the value of <control-item> determines the number of iterations. If the <control-item> is an <item-name>, its type is as specified in its <declaration>, and that <item-name> may be used for purposes other than loop control before and after the loop. After execution of the loop concludes, the value of the <item-name> is the last value it received in the <loop-statement>. If the <control-item> is a <letter>, the <for-clause> constitutes an implicit declaration of the <control-item>, and its value is inaccessible prior to the start of the <loop-statement> and after the <loop-statement> concludes. Its type is that of the <initial-value>. Its scope is the <loop-statement> itself; hence, another <loop-statement> may use the same <letter> as a <control-item> (except as prohibited in Constraints) and no conflict will result.

The actions of the <loop-statement> with a <for-clause> are as specified by the following algorithm:

Step 1: The <initial-value> is evaluated and assigned to the <control-item>.

Step 2: The <boolean-formula> in the <while-phrase> (if present) is evaluated. If it is FALSE, execution of the <loop-statement> concludes.

Step 3: The <controlled-statement> is executed.

Step 4: The formula in the <by-or-then-phrase> (if present) is evaluated. The value for the <by-formula> (if present) is added to the <control-item>. The value of the <then-formula> (if present) is assigned to the <control-item>. Execution continues at Step 2.

The <control-item> may be used in <formulas> in the <control-clause> and in the <controlled-statement>.

Execution of the <loop-statement> concludes if control is passed to another statement by means of a GOTO, RETURN, EXIT, STOP, or ABORT statement.

Constraints:

If the <control-item> is a <letter>, it must not be used in the <controlled-statement> or <control-clause> in any context in which its value can be altered (e.g., as an <actual-output-parameter> or as a target in an <assignment-statement>). If the <control-item> is an <item-name>, assignments to it in the <controlled-statement> are not prohibited, but will result in a warning message.

A <label> in a <controlled-statement> cannot be used as the <statement-name> in a <goto-statement> or <abort-phrase> that is outside the <controlled-

statement> or as an <actual-input-parameter> in a subroutine invocation that is outside the <controlled-statement>.

The <initial-value>, <by-formula> and <then-formula> must be equivalent or implicitly convertible to the type of the <control-item> (see Section 7.0). Further, the sum of the <by-formula> and the <control-item> must be equivalent or implicitly convertible to the type of the <control-item>. The <initial-value> cannot be of type table.

The <by-formula> (if present) must have type and value such that it may be legally added to the <control-item> according to the rules of Sections 5.1.1, 5.1.2, and 5.1.3.

If the <control-item> is a <control-letter>, the <initial-value> must not be a <status-constant> that belongs to more than one type (unless the <status-constant> is disambiguated by an explicit conversion -- see Section 7).

The <control-letter> in a <loop-statement> may not be the same as the <control-letter> of any enclosing <loop-statement>.

A <bit-formula> cannot be implicitly converted to the <boolean-formula> in a <while-clause> or <while-phrase>.

Notes:

There is a danger that a program may inadvertently attempt to set the <control-item> to a value invalid for its type. Here are some examples of such program errors:

1.
```
ITEM II S 15;

FOR II: 1 BY 1 WHILE II <= MAXINT(15);
```

Presumably the intention is to execute the loop MAXINT(15) times; at the end of this execution of the loop, II will have the value MAXINT(15). An attempt to add one to II will then produce an undefined result. The consequences are implementation-dependent.

2.
```
TYPE LETTERS STATUS 2 (V(A), V(B), V(C), V(D));

FOR I : FIRST (LETTERS) THEN NEXT (I,1)

WHILE I < = LAST (LETTERS);
```

At the end of the fourth iteration of the loop, I has the value V(D). An attempt is then made to evaluate the expression NEXT (V(D),1), which is invalid. The consequences are implementation-dependent.

4.3 IF STATEMENTS

Syntax:

<if-statement>	::=	IF <boolean-formula> ; <conditional-statement> [<else-clause>]	(5.2.2)
<conditional-statement>	::=	<statement>	(4.0)
<else-clause>	::=	ELSE <statement>	(4.0)

Semantics:

An <if-statement> provides for conditional execution of a statement depending on the value of its <boolean-formula>.

If the value of the <boolean-formula> is TRUE, the <conditional-statement> is executed and the <statement> in the ELSE clause (if any) is not executed.

If the value of the <boolean-formula> is FALSE, the <statement> in the <else-clause> (if present) is executed rather than the <conditional-statement>. In the event of nested <if-statements>, an ELSE associates with the innermost unmatched IF.

If the <boolean-formula> has a value that is known at compile time, conditional compilation (see Section 1.3.1) will occur.

Constraints:

<Labels> throughout a scope must be unique, even if portions of the text within the scope are unselected as a result of conditional compilation.

A <bit-formula> cannot be implicitly converted to the <boolean-formula> in an <if-statement>.

Note:

<Labels> in the <conditional-statement> and in the <else-clause> are in the same scope as the <if-statement> itself.

4.4 CASE STATEMENTS

Syntax:

<case-statement>	::=	CASE <case-selector-formula> ; BEGIN <case-body> [<label>...] END	(4.0)

```
<case-selector-formula>  ::=  <integer-formula>                  (5.1.1)
                         |    <bit-formula>                      (5.2)
                         |    <character-formula>                (5.3)
                         |    <status-formula>                   (5.4)
<case-body>              ::=  <case-alternative>...
<case-alternative>       ::=  <case-index-group>
                              <statement>                       (4.0)
                              [FALLTHRU]
                         |    <default-option>
<default-option>         ::=  ( DEFAULT ) :
                              <statement>                       (4.0)
                              [FALLTHRU]
<case-index-group>       ::=  ( <case-index>,... ) :
<case-index>             ::=  <compile-time-integer-formula>    (5.1.1)
                         |    <compile-time-bit-formula>        (5.1.2)
                         |    <compile-time-character-          (5.3)
                                formula>
                         |    <compile-time-status-formula>     (5.4)
                         |    <lower-bound> :                   (2.1.2.1)
                              <upper-bound>                     (2.1.2.1)
```

Semantics:

Whereas an <if-statement> provides for the optional execution of either of two statements, a <case-statement> provides for a choice of executing one or more of a number of statements. (The possible choices are represented by the various <case-alternatives>).

The particular <case-alternative> is selected according to the value of <case-selector-formula>. Several values of the <case-selector-formula> may select the same <case-alternative>.

With the exception of the <default-option>, each <case-alternative> is headed by a <case-index-group> that designates the possible values of the <case-selector-formula> that after being implicitly converted (if necessary) to the type of the <case-selector-formula>, cause that particular <case-alternative> to be selected for execution. Each <case-index> can designate either a single

value or, for integer and status selector types, a closed range of values bounded by <lower-bound> and <upper-bound>.

If the value of the <case-selector-formula> does not correspond to a <case-index> value, the <statement> in the <default-option> is executed.

If FALLTHRU is not present after a selected <statement>, execution of the <case-statement> concludes after that <statement> is executed. If FALLTHRU is present after the selected <statement>, the <statement> in the textually-succeeding <case-alternative> is then executed. Control continues to "fall through" to subsequent <case-alternatives>, until a case-alternative with no FALLTHRU is executed or until the END of the <case-statement> has been reached.

If the value of the <case-selector-formula> is known at compile time, conditional compilation (see Section 1.3.1) will occur for all unselected alternatives that cannot be reached via FALLTHRU semantics.

Constraints:

No two <case-alternatives> within the same <case-statement> can be associated with identical <case-index> values.

If a <default-option> is not present, the value of the <case-selector-formula> must be represented by a <case-index>.

The types of each formula in a <case-index> must match or be implicitly convertible to that of the <case-selector-formula> according to the rules given in Section 7.0.

If the <case-selector-formula> is a <status-formula>, a <case-index> specifying lower and upper bounds is legal only if the status-type has the default representation (see Section 2.1.1.6).

The <upper-bound> in a <case-index> must be greater than or equal to the <lower-bound>.

Within a <case-statement>, at most one <default-option> may be used as a <case-alternative>.

Note:

<Labels> in the <default-option> and in the <case-alternatives> are in the same scope as the <case-statement> itself. Consequently, control can be transferred into or between case statements.

4.5 PROCEDURE CALL STATEMENTS

Syntax:

```
<procedure-call-
    statement>              ::=  <user-defined-procedure-call>

                              |  <machine-specific-procedure-call>

<user-defined-procedure-
    call>                  ::=  <procedure-name>                    (3.1)
                                [<actual-parameter-list>]
                                [<abort-phrase>]  ;

<actual-parameter-list>    ::=  (  [<actual-input-parameter>,...]
                                [  :  <actual-output-parameter>,...  ]  )

<actual-input-parameter>   ::=  <formula>                           (5.0)

                              |  <statement-name>                   (4.0)

                              |  <function-name>                    (3.2)

                              |  <procedure-name>                   (3.1)

                              |  <block-reference>                  (6.1)

<actual-output-
    parameter>             ::=  <variable>                          (6.1)

                              |  <block-reference>                  (6.1)

<abort-phrase>             ::=  ABORT  <statement-name>              (4.0)

<machine-specific-
    procedure-call>        ::=  <procedure-name>                    (3.1)
                                [<actual-parameter-list>]  ;
```

Semantics:

A <procedure-call-statement> causes invocation of a procedure and the association of formal parameters with actual parameters according to the rules given in Section 3.3.

A <user-defined-procedure-call> causes invocation of a procedure defined in a <procedure-definition>. The <abort-phrase> is for use in connection with <abort-statements>. Its semantics are explained in Section 4.10.

A <machine-specific-procedure-call> causes invocation of a machine-specific procedure (see Section 3.5).

Constraints:

Actual parameters in the <procedure-call-statement> must match the formal parameters of the called procedure in number, kind, and parameter list position, according to the rules given in Section 3.3.

The <statement-name> in an <abort-phrase> or <actual-input-parameter> must conform to the constraints specified in Section 4.7 for <statement-names> used in <goto-statements>.

4.6 RETURN STATEMENTS

Syntax:

<return-statement> ::= RETURN ;

Semantics:

The effect of a <return-statement> is to terminate the execution of a subroutine, set any parameters that have result or value-result semantics, and return control to the point following the invocation of the subroutine. If the <return-statement> is in a <function-body>, the current value of the <function-name> becomes the value of the function call.

If the subroutine containing the <return-statement> is nested within any enclosing subroutines, only the innermost subroutine is terminated.

Constraint:

The <return-statement> can appear only within the body of a subroutine.

4.7 GOTO STATEMENTS

Syntax:

<goto-statement> ::= GOTO <statement-name> ; (4.0)

Semantics:

A <goto-statement> causes control to be transferred to the statement named by the specified <statement-name>.

When the <statement-name> is a formal statement-name parameter, the effect of a <goto-statement> is equivalent to returning from the current subroutine

invocation without setting result or value-result parameters and then executing a <goto-statement> at the point of the subroutine's invocation.

Constraints:

The <statement-name> must be known in the scope in which the <goto-statement> appears. Further, the <statement-name> must not be the <label> of a statement that is in an enclosing subroutine or in another module. It cannot be the <label> of a statement in a <controlled-statement> unless the <goto-statement> is itself within that same <controlled-statement>.

4.8 EXIT STATEMENTS

Syntax:

<exit-statement> ::= EXIT ;

Semantics:

An <exit-statement> causes execution of the immediately enclosing <loop-statement> to terminate. Its effect is the same as a GOTO statement that transfers control out of the <controlled-statement> to the point following the end of the <loop-statement>.

Constraint:

The <exit-statement> can appear only in a <controlled-statement>.

4.9 STOP STATEMENTS

Syntax:

<stop-statement> ::= STOP [<integer-formula>] ; (5.1.1)

Semantics:

A <stop-statement> causes execution of the <complete-program> to terminate. If a <stop-statement> is executed within a <subroutine-body>, the value-result <actual-output-parameters> of any subroutine whose call is still active will not be set.

The value of the optional <integer-formula> in a <stop-statement> is made available to the environment in which the J73 program is executing, where its semantics are implementation-dependent. Absence of an <integer-formula> implies the value is not determined.

Constraint:

The range of legal values of the <integer-formula> is MINSTOP through MAXSTOP.

4.10 ABORT STATEMENTS

Syntax:

<abort-statement> ::= ABORT ;

Semantics:

When an <abort-statement> is executed, control passes to the statement named in the <abort-phrase> of the most recently executed, currently active <procedure-call-statement> that has an <abort-phrase>. All intervening subroutine invocations are terminated, and value-result parameters of such subroutines are not set. If there is no currently-active <procedure-call-statement> that has an <abort-phrase>, the effect of the <abort-statement> is the same as STOP.

5.0 FORMULAS

Syntax:

<formula>	::=	<numeric-formula>	(5.1)
	|	<bit-formula>	(5.2)
	|	<character-formula>	(5.3)
	|	<status-formula>	(5.4)
	|	<pointer-formula>	(5.5)
	|	<table-formula>	(5.6)
<compile-time-formula>	::=	<compile-time-numeric-formula>	(5.1)
	|	<compile-time-bit-formula>	(5.2)
	|	<compile-time-character-formula>	(5.3)
	|	<compile-time-status-formula>	(5.4)
	|	<compile-time-pointer-formula>	(5.5)

Semantics:

<Formulas> represent values. Each <formula> has associated with it a type class and appropriate attributes.

A <compile-time-formula> is a <formula> whose value is computed and used at compile time.

All compile-time computations are performed using the range and precision parameters of the target machine.

The following constructions yield values at compile time.

1. Data declared in <constant-item-declarations>, except for constant items whose type class is pointer.

2. The functions LBOUND, FIRST, and LAST, regardless of their arguments; the functions UBOUND and NENT, provided their argument is not a table with * dimensions; the functions NEXT, BIT, BYTE, SHIFTL, SHIFTR, ABS, and SGN, provided their arguments are known at compile time; the function NWDSEN, provided its argument does not contain a reference to a name whose declaration is not completed prior to the point at which the function appears; the functions BITSIZE, BYTESIZE, and WORDSIZE, provided (1) their arguments are type names or have sizes that are known at compile time, (2) their arguments do not contain references to names whose

declarations are not completed prior to the points at which the functions appear, and (3) their arguments are not blocks and are not tables with * dimensions.

3. All operator-operand combinations other than dereferencing, indexing, and assignment, provided the operands have values that are known at compile time.

4. All type conversions except REP, provided the value of the <formula> being converted is known at compile time.

5. All machine parameters.

6. All <status-constants>.

7. All <literals>.

The following values are not known at compile time:

1. Constant items whose type class is pointer.

2. Constant tables and their components.

3. All data declared without the word CONSTANT.

4. The LOC function, regardless of its argument; the functions UBOUND and NENT, if their argument is a table with * dimensions; the functions NEXT, BIT, BYTE, SHIFTL, SHIFTR, ABS, and SGN, if they have one or more arguments whose values are not known at compile time; the function NWDSEN, if its argument is a name whose declaration is not completed prior to the point at which the function appears; the functions BITSIZE, BYTESIZE, and WORDSIZE, if (1) their arguments are not type names and have sizes that are not known at compile time, (2) their arguments contain references to names whose declarations are not completed prior to the points at which the functions appear, or (3) their arguments are blocks or tables with * dimensions.

5. All operator-operand combinations that have one or more operands whose values are not known at compile time.

6. The REP conversion.

7. Any value arrived at via a <statement>.

8. Dereferenced or subscripted values.

Any value known at compile time may also be used as a run-time value.

5.1 NUMERIC FORMULAS

Syntax:

```
<numeric-formula>                  ::= <integer-formula>                  (5.1.1)
                                     | <floating-formula>                 (5.1.2)
                                     | <fixed-formula>                    (5.1.3)
<compile-time-numeric-formula>::= <compile-time-integer-formula>  (5.1.1)
                                     | <compile-time-floating-formula>(5.1.2)
                                     | <compile-time-fixed-formula>     (5.1.3)
```

Semantics:

A <numeric-formula> represents a numeric value.

A <compile-time-numeric-formula> represents a numeric value that is known at compile time (see Section 5.0).

5.1.1 INTEGER FORMULAS

Syntax:

```
<integer-formula>          ::= [<sign>] <integer-term>          (8.3.1)
                             | <integer-formula>
                               <plus-or-minus>                  (8.2.3)
                               <integer-term>

<integer-term>             ::= <integer-factor>
                             | <integer-term>
                               <multiply-divide-or-mod>         (8.2.3)
                               <integer-factor>

<integer-factor>           ::= <integer-primary>
                             | <integer-factor>  **
                               <integer-primary>

<integer-primary>          ::= <integer-literal>                (8.3.1)
                             | <integer-machine-
                                 parameter>                     (1.4)
                             | <integer-variable>
```

		¦ <named-integer-constant>	
		¦ <integer-function-call>	
		¦ (<integer-formula>)	
		¦ <integer-conversion>	(7.0)
		(<formula>)	(5.0)
<integer-variable>	::=	<variable>	(6.1)
<named-integer-constant>	::=	<named-constant>	(6.2)
<integer-function-call>	::=	<function-call>	(6.3)
<compile-time-integer-formula>	::=	<integer-formula>	

Semantics:

An <integer-formula> represents a value whose type class is integer, i.e., S or U.

The integer operators are +, -, *, /, MOD, and **, which denote addition, subtraction, multiplication, division, modulus, and exponentiation, respectively.

The type of a formula composed of an integer operator and two operands is S NN-1, where NN is the actual number of bits that would be supplied by the implementation for a signed integer <item-declaration> whose size attribute is the larger of the size attributes of the two operands. The type of an <integer-formula> consisting of a <sign> and an <integer-term> is S NN-1, where NN is the actual number of bits that would be supplied for a signed integer <item-declaration> whose size attribute is that of the <integer-term>.

The quotient of an <integer-term> AA and a non-zero <integer-factor> BB, expressed as AA/BB, is defined as follows: Let QABS be the greatest integer not exceeding the absolute value of the mathematical quotient (AA divided by BB). The <integer-term> AA/BB has the value zero if AA equals zero, QABS if AA and BB have the same sign, and -(QABS) otherwise.

The modulus of two integers, AA MOD BB, is equivalent to AA - (AA/BB)*BB.

The value produced by integer exponentiation to a positive power is the same as that produced by repeated multiplication.

The value produced by integer exponentiation to a negative power is 1 / (base ** abs (power)) and in most cases is zero.

The value produced by integer exponentiation to the zero power is one.

Constraints:

The value of an <integer-formula> with size attribute SS must lie in the range MININT(SS) through MAXINT(SS).

An <integer-variable>, <named-integer-constant>, or <integer-function-call> must be an integer (S or U) type.

A <compile-time-integer-formula> must be an <integer-formula> whose value is known at compile-time (see Section 5.0).

The right operand of / and MOD must be non-zero.

Note:

R, T, and Z used in an explicit conversion (see Section 7.0) do not affect the value of integer division.

5.1.2 FLOATING FORMULAS

Syntax:

```
<floating-formula>          ::= [<sign>] <floating-term>                (8.3.1)

                              | <floating-formula>
                                <plus-or-minus>                         (8.2.3)
                                <floating-term>

<floating-term>             ::= <floating-factor>

                              | <floating-term>
                                <multiply-or-divide>                    (8.2.3)
                                <floating-factor>

<floating-factor>           ::= <floating-primary>

                              | <floating-factor>
                                ** <floating-primary>

                              | <floating-factor>
                                ** <integer-primary>                    (5.1.1)

<floating-primary>          ::= <floating-literal>                      (8.3.1)

                              | <floating-machine-parameter>            (1.4)

                              | <floating-variable>

                              | <named-floating-constant>

                              | <floating-function-call>
```

		¦ (<floating-formula>)	
		¦ <floating-conversion>	(7.0)
		(<formula>)	(5.0)
<floating-variable>	::=	<variable>	(6.1)
<named-floating-constant>	::=	<named-constant>	(6.2)
<floating-function-call>	::=	<function-call>	(6.3)
<compile-time-floating-formula>	::=	<floating-formula>	

Semantics:

A <floating-formula> represents a value whose type class is float.

The floating operators are +, -, *, /, and **, which denote addition, subtraction, multiplication, division, and exponentiation respectively. In exponentiation with a <floating-factor>, a floating value is produced in all cases.

The precision attribute of a <floating-formula> is that of the formula's most precise floating operand. The operand of a <floating-conversion> is first computed according to the default rules, and then converted to the specified floating type (see Section 7.0).

For floating exponentiations whose right operand is an <integer-primary>, the result is -(ABS (left operand) ** right operand) if left operand is negative and right operand is odd; (ABS (left operand) ** right operand) in all other cases.

Constraints:

The value of a <floating-formula> with precision PP must lie in the range FLOATUNDERFLOW (II) through MAXFLOAT (II) or the range MINFLOAT (II) through -FLOATUNDERFLOW (II) or be zero, where II = IMPLFLOATPRECISION(PP).

A <floating-variable>, <named-floating-constant>, or <floating-function-call> must be a floating type.

A <compile-time-floating-formula> must be a <floating-formula> whose value is known at compile time (see Section 5.0).

For exponentiations where the right operand is a <floating-primary>, the left operand must not be negative.

Exponentiation of an integer base to a floating power cannot be performed. Either the base must be converted to floating or the power must be converted to integer.

The divisor must be non-zero.

Note:

The round or truncate attribute associated with variables or constant names does not affect the computation of floating formula results. Floating formulas are evaluated in an implementation-dependent manner with respect to how exact results are approximated to the implemented precision.

5.1.3 FIXED FORMULAS

Syntax:

<fixed-formula>	::=	[<sign>] <fixed-term>	(8.3.1)
	¦	<fixed-formula> <plus-or-minus>	(8.2.3)
		<fixed-term>	
<fixed-term>	::=	<fixed-factor>	
	¦	<fixed-term> *	
		<fixed-factor>	
	¦	<integer-term> *	(5.1.1)
		<fixed-factor>	
	¦	<fixed-term>	
		<multiply-or-divide>	(8.2.3)
		<integer-factor>	(5.1.1)
<fixed-factor>	::=	<fixed-literal>	(8.3.1)
	¦	<fixed-machine-parameter>	(1.4)
	¦	<fixed-variable>	
	¦	<named-fixed-constant>	
	¦	<fixed-function-call>	
	¦	(<fixed-formula>)	
	¦	<fixed-conversion>	(7.0)
		(<fixed-term> /	
		<fixed-factor>)	
	¦	<fixed-conversion>	(7.0)
		(<integer-term> /	
		<fixed-factor>)	

		| <fixed-conversion>	(7.0)
		(<formula>)	(5.0)
<fixed-variable>	::=	<variable>	(6.1)
<named-fixed-constant>	::=	<named-constant>	(6.2)
<fixed-function-call>	::=	<function-call>	(6.3)
<compile-time-fixed-formula>	::=	<fixed-formula>	

Semantics:

A <fixed-formula> represents a fixed point value.

The fixed point operators are +, -, *, and /, which denote addition, subtraction, multiplication, and division, respectively. The rules specifying the result type of these operators guarantee that, in general, exact results are produced. The specific rules are given below for each operator. In these rules, Sn, Fn, and Pn refer to the scale, fraction part, and precision of an operand or result and n is 1, 2, or R to indicate the first operand, second operand, or result, respectively.

For addition and subtraction, the default type of the result is:

```
SR = S1 = S2
FR = Max (F1,F2)
PR = Max (P1,P2)
```

For multiplication, there are two cases:

1. When one operand is an integer, the result scale and precision are those produced by successive addition, i.e.,

```
SR = Sa
FR = Fa
PR = Pa
```

   where Sa, Fa, and Pa represent the scale, fraction, and precision values of the fixed point operand.

2. When both operands are fixed point types, the type of the result is:

```
SR = S1 + S2
FR = F1 + F2
PR = P1 + P2
```

If PR is larger than MAXFIXEDPRECISION or if SR does not lie in the range - 127 through + 127, then the product must be explicitly converted to a valid fixed point scale and precision (see Section 7.0).

For division, there are also two cases:

1. When dividing a fixed point value by an integer, the scale and precision of the result are the scale and precision of the numerator. Truncation will be toward zero.

2. When both operands are fixed point values or when an integer is divided by a fixed point value, the result is exact and must be explicitly converted to a programmer specified scale and precision (see Section 7.0).

The default result type of a <fixed-formula> containing a <sign> as a prefix operator is the type of the operand.

The result type of a <fixed-factor> that is a <fixed-variable>, <named-fixed-constant>, or <fixed-function-call> is the type specified in their respective variable, constant, or function declarations.

The type of a <fixed-literal> is contextually determined (see Section 8.3.1).

The result type of a <fixed-formula> enclosed in parentheses is the type of the enclosed <fixed-formula>.

The result type of a <fixed-factor> containing a <fixed-conversion> is the type specified by the <fixed-conversion>. If the operand of the <fixed-conversion> is a <fixed-term> or <fixed-formula>, the infix or unary operator is evaluated exactly, and the mathematically-defined result is converted to the specified fixed type.

Constraints:

Except for the operand of a <fixed-conversion>, the value of a <fixed-formula> whose scale is SS and whose fraction attribute is FF must lie in the range MINFIXED(SS,PP-SS) through MAXFIXED(SS,PP-SS), where PP = IMPLFIXEDPRECISION (SS,FF).

A <fixed-variable>, <named-fixed-constant>, and <fixed-function-call> must have been declared as fixed types.

Operands of fixed point addition or subtraction must have identical scales.

A <compile-time-fixed-formula> must be a <fixed-formula> whose value is known at compile time (see Section 5.0).

The divisor must be non-zero.

Note:

MOD and ** are not defined for fixed point operands.

5.2 BIT FORMULAS

Syntax:

```
<bit-formula>              ::= <logical-operand>
                               [<logical-continuation>]
                             | NOT <logical-operand>

<logical-operand>          ::= <bit-primary>
                             | <relational-expression>      (5.2.1)

<bit-primary>              ::= <bit-literal>               (8.3.2)
                             | <boolean-literal>           (8.3.3)
                             | <bit-variable>
                             | <named-bit-constant>
                             | <bit-function-call>
                             | ( <bit-formula> )
                             | <bit-conversion>            (7.0)
                               ( <formula> )               (5.0)

<logical-continuation>     ::= <and-continuation>...
                             | <or-continuation>...
                             | <xor-continuation>...
                             | <eqv-continuation>...

<and-continuation>         ::= AND <logical-operand>

<or-continuation>          ::= OR <logical operand>

<xor-continuation>         ::= XOR <logical-operand>

<eqv-continuation>         ::= EQV <logical-operand>

<bit-variable>             ::= <variable>                  (6.1)

<named-bit-constant>       ::= <named-constant>            (6.2)
```

<bit-function-call> ::= <function-call> (6.3)

<compile-time-bit-formula> ::= <bit-formula>

Semantics:

A <bit-formula> represents a value whose type class is bit. Its size is the number of bits comprising its value.

If the <bit-formula> is composed of <logical-operands> and one or more of the logical operators AND, OR, XOR, and EQV, the size of the result is the size of the longest operand. Shorter operands are padded on the left with zeros as necessary. Note that the syntax requires explicit parentheses for all <bit-formulas> containing two or more of these operators, unless the operators are identical.

NOT produces a value that is the logical complement of its operand. AND, OR (inclusive or), XOR (exclusive or), and EQV (equivalence) perform their usual logical function on their two operands on a bit-by-bit basis. If both operands have a size of one bit and the value of the left operand is such that the result of the operator can be determined, evaluation is "short-circuited", i.e., the right operand will not be evaluated and need only satisfy semantic constraints that can always, even in the most general case, be verified without evaluating the operand (e.g., the operand need not satisfy the division by zero constraint if it is not evaluated).

Constraints:

A <bit-variable> must be a <variable> whose type class is bit.

A <named-bit-constant> must be a <named-constant> whose type class is bit.

A <bit-function-call> must be a <function-call> whose result value is bit.

A <compile-time-bit-formula> must be a <bit-formula> whose value is known at compile time (see Section 5.0).

5.2.1 RELATIONAL EXPRESSIONS

Syntax:

<relational-expression> ::= <integer-formula> (5.1.1)
<relational-operator> (8.2.3)
<integer-formula> (5.1.1)

| <floating-formula> (5.1.2)
<relational-operator> (8.2.3)
<floating-formula> (5.1.2)

¦	<fixed-formula>	(5.1.3)
	<relational-operator>	(8.2.3)
	<fixed-formula>	(5.1.3)
¦	<character-formula>	(5.3)
	<relational-operator>	(8.2.3)
	<character-formula>	(5.3)
¦	<status-formula>	(5.4)
	<relational-operator>	(8.2.3)
	<status-formula>	(5.4)
¦	<bit-primary>	(5.2)
	<equal-or-not-equal-operator>	(8.2.3)
	<bit-primary>	(5.2)
¦	<pointer-formula>	(5.5)
	<relational-operator>	(8.2.3)
	<pointer-formula>	(5.5)

Semantics:

A <relational-expression> represents a value obtained by comparing two formulas using a <relational-operator>. Its type class is B and its size is one bit.

The relational operators, = (equal), <> (not equal), < (less than), > (greater than), <= (less than or equal), and >= (greater than or equal), carry their usual meanings.

Character comparisons will be made on the basis of the collating sequence of the character set used in a given implementation.

Status comparisons will be made on the basis of the representation of the status values.

Pointer comparisons will be made on a target-machine-dependent basis.

For bit and character operands, the shorter will be implicitly converted to the type of the longer as described in Section 7.0.

Constraints:

When both operands are <status-constants>, at least one must be unambiguously associated with a single status type.

When the two operands are <status-formulas>, their types must be identical.

When the two operands are <pointer-formulas>, their types must be identical or one must be an untyped pointer.

When both operands are <fixed-formulas>, there must exist a type to which both operands are implicitly convertible.

5.2.2 BOOLEAN FORMULAS

Syntax:

<boolean-formula> ::= <bit-formula> (5.2)

Semantics:

A <boolean-formula> is a <bit-formula> whose size is one bit. It has the value TRUE if the value of the bit is one and FALSE otherwise.

Constraints:

In contexts syntactically requiring a <boolean-formula> (<if-statements>, <while-clauses>, <while-phrases>, and <trace-controls>), a <bit-formula> cannot be implicitly converted to a <boolean-formula>.

5.3 CHARACTER FORMULAS

Syntax:

<character-formula> ::= <character-literal> (8.3.4)
| <character-variable>
| <named-character-constant>
| <character-function-call>
| (<character-formula>)
| <character-conversion> (7.0)
(<formula>) (5.0)

<character-variable> ::= <variable> (6.1)

<named-character-constant> ::= <named-constant> (6.2)

<character-function-call> ::= <function-call> (6.3)

<compile-time-character-formula> ::= <character-formula>

Semantics:

A <character-formula> represents a value whose type class is character. Its size is the number of bytes comprising its value.

Constraints:

A <character-variable> must be a <variable> whose type class is character.

A <named-character-constant> must be a <named-constant> whose type class is character.

A <character-function-call> must be a <function-call> whose result value is character.

A <compile-time-character-formula> must be a <character-formula> whose value is known at compile time (see Section 5.0).

5.4 STATUS FORMULAS

Syntax:

<status-formula>	::=	<status-constant>	(2.1.1.6)
	¦	<status-variable>	
	¦	<named-status-constant>	
	¦	<status-function-call>	
	¦	(<status-formula>)	
	¦	<status-conversion>	(7.0)
		(<formula>)	(5.0)
<status-variable>	::=	<variable>	(6.1)
<named-status-constant>	::=	<named-constant>	(6.2)
<status-function-call>	::=	<function-call>	(6.3)
<compile-time-status-formula>	::=	<status-formula>	

Semantics:

A <status-formula> represents a value whose type class is status.

Constraints:

A <status-variable> must be a <variable> whose type class is status.

A <named-status-constant> must be a <named-constant> whose type class is status.

A <status-function-call> must be a <function-call> whose result value is status.

A <compile-time-status-formula> must be a <status-formula> whose value is known at compile time (see Section 5.0).

5.5 POINTER FORMULAS

Syntax:

<pointer-formula>	::=	<pointer-literal>	(8.3.5)
	¦	<pointer-variable>	
	¦	<named-pointer-constant>	
	¦	<pointer-function-call>	
	¦	(<pointer-formula>)	
	¦	<pointer-conversion>	(7.0)
		(<formula>)	(5.0)
<pointer-variable>	::=	<variable>	(6.1)
<named-pointer-constant>	::=	<named-constant>	(6.2)
<pointer-function-call>	::=	<function-call>	(6.3)
<compile-time-pointer-formula>	::=	<pointer-formula>	

Semantics:

A <pointer-formula> represents a value whose type class is pointer.

Constraints:

A <pointer-variable> must be a <variable> whose type class is pointer.

A <named-pointer-constant> must be a <named-constant> whose type class is pointer.

A <pointer-function-call> must be a <function-call> whose result value is pointer.

A <compile-time-pointer-formula> must be a <pointer-formula> whose value is known at compile time (see Section 5.0).

5.6 TABLE FORMULAS

Syntax:

<table-formula>	::=	<table-variable>

		¦ <named-table-constant>	
		¦ (<table-formula>)	
		¦ <table-conversion>	(7.0)
		(<formula>)	(5.0)
<table-variable>	::=	<variable>	(6.1)
<named-table-constant>	::=	<named-constant>	(6.2)

Semantics:

A <table-formula> represents a value whose type class is table.

Constraints:

A <table-variable> must be a <variable> whose type class is table.

A <named-table-constant> must be a <named-constant> whose type class is table.

6.0 DATA REFERENCES

6.1 VARIABLE AND BLOCK REFERENCES

Syntax:

<variable>	::=	<named-variable>	
	¦	<bit-function-variable>	
	¦	<byte-function-variable>	
	¦	<rep-function-variable>	
	¦	<function-name>	(3.2)
<named-variable>	::=	<item>	
	¦	<table>	
	¦	<table-item>	
	¦	<table-entry>	
	¦	<block-item>	
	¦	<block-table>	
	¦	<block-table-item>	
	¦	<block-table-entry>	
<item>	::=	<item-name>	(2.1.1)
	¦	<item-dereference>	
<table>	::=	<table-name>	(2.1.2)
	¦	<table-dereference>	
<table-item>	::=	<table-item-name> [<subscript>] [<table-dereference>]	(2.1.2.3)
<table-entry>	::=	<table-name> <subscript>	(2.1.2)
	¦	<table-dereference> <subscript>	
<block-item>	::=	<item-name> [<block-dereference>]	(2.1.1)

```
<block-table>              ::= <table-name>                          (2.1.2)
                               [<block-dereference>]

<block-table-item>         ::= <table-item-name>                     (2.1.2.3)
                               [<subscript>]
                               [<block-dereference>]

<block-table-entry>        ::= <table-name>                          (2.1.2)
                               <subscript>
                               [<block-dereference>]

<block-dereference>        ::= <dereference>

<item-dereference>         ::= <dereference>

<table-dereference>        ::= <dereference>

<dereference>              ::= @ <pointer-item-name>

                             | @ ( <pointer-formula> )               (5.5)

<pointer-item-name>        ::= <item-name>                           (2.1.1)

                             | <table-item-name>                     (2.1.2.3)

                             | <constant-item-name>                  (2.1.3)

<subscript>                ::= ( <index>,... )                       (5.1.1)

<index>                    ::= <integer-formula>                     (5.1.1)

                             | <status-formula>                      (5.4)

<bit-function-variable>    ::= BIT ( <bit-variable> ,                (5.2)
                               <fbit> , <nbit> )                     (6.3.3)

<byte-function-variable>   ::= BYTE (
                               <character-variable> ,                (5.3)
                               <fbyte> , <nbyte> )                   (6.3.4)

<rep-function-variable>    ::= <rep-conversion>                      (7.0)
                               ( <named-variable> )

<block-reference>          ::= <block-name>                          (2.1.4)

                             | <block-dereference>

                             | <block-name>                          (2.1.4)
                               <block-dereference>
```

Semantics:

A <variable> designates a data object whose value can be changed by assignment. A <named-variable> designates a data object whose value can be used in a formula and changed by assignment. A <dereference> designates the data object whose address is contained in the <pointer-item-name> or <pointer-formula> of the <dereference>.

An <item> variable designates either an object declared in an <item-declaration> or an object pointed to by a typed pointer whose <type-name> attribute is an item type. In the latter case the item is referenced with an <item-dereference> (i.e., the pointer is dereferenced to obtain the item).

A <table> variable designates either an object declared in a <table-declaration> or an object pointed to by a typed pointer whose <type-name> attribute is a table type. In the latter case the table is referenced with a <table-dereference> (i.e., the pointer is dereferenced to obtain the table). The type class of a <table> is table.

A <table-item> variable designates an item component of a table. If the table is dimensioned, the subscript indicates from which entry the item is to be obtained. If <table-item-name> was declared in a <table-type-declaration> (rather than a <table-item-declaration>) the <table-dereference> references the particular table from which the item is to be obtained.

A <table-entry> variable designates an entry in a dimensioned table. The table is referenced either with a <table-name> or with a <table-dereference>.

The type class of a <table-entry> is table for entries declared with an <ordinary-table-body>, <specified-table-body>, or <table-type-name>, and otherwise is the type specified by the underlying <item-type-description>. (Note that <table-entry> is syntactically a subscripted <table-name> or <table-dereference>.)

If the type class of a particular <table-entry> is not table, any operation or intrinsic function except LOC, NWDSEN, and REP applied to that entry is interpreted as applying to the item whose type class and attributes are given by the underlying <item-type-description>. LOC, NWDSEN, and REP are interpreted as applying to the entire physical space occupied by the object, including filler bits preceding or following the item.

A <block-item> variable designates an item component of a block. If the <item-name> was declared in a <block-type-declaration>, the <block-dereference> references the particular block from which the item is to be obtained.

A <block-table> variable designates a table component of a block. If the <table-name> was declared in a <block-type-declaration>, the <block-dereference> references the particular block from which the table is to be obtained.

A <block-table-item> variable designates an item component of a table which is itself a component of a block. If the table is dimensioned, the subscript indicates from which entry the item is to be obtained. If the <table-item-name>

was declared in a <block-type-declaration>, the <block-dereference> references the particular block from which the item is to be obtained. (Note that if the <table-item-name> was declared in a <table-type-declaration>, it cannot be obtained as a <block-table-item> variable but must be obtained as a <table-item> variable with a <table-dereference>.)

A <block-table-entry> variable designates an entry in a dimensioned table which is contained in a block. If the <table-name> was declared in a <block-type-declaration>, the <block-dereference> references the particular block from which the table entry is to be obtained.

A <bit-function-variable> is the use of the BIT function in an assignment context (i.e., the target of an <assignment-statement> or an <actual-output-parameter>) to designate that a specified substring of the <bit-variable> is to be used as a variable. <Fbit> indicates the starting bit and <nbit> indicates the size of the substring. Bits are numbered from the left beginning with zero.

A <byte-function-variable> is the use of the BYTE function in an assignment context (i.e., a target of an <assignment-statement> or an <actual-output-parameter>) to designate that a specified substring of the <character-variable> is to be used as a variable. <Fbyte> indicates the starting character and <nbyte> indicates the size of the substring. Characters are numbered from the left beginning with zero.

A <rep-function-variable> is the use of the <rep-conversion> in an assignment context (i.e., the target of an <assignment-statement> or an <actual-output-parameter>) to designate that the <named-variable> is to be treated as a bit string variable whose size is the number of bits of storage actually occupied by the <named-variable>.

A <block-reference> designates either an object declared in a <block-declaration> or an object whose attributes are specified in a <block-type-declaration> and whose location is specified via a <block-dereference>. The <block-reference> is specified by any of the following:

- <Block-dereference> - to access a block by specifying its <type-name> and location (<type-name> is implied by the dereference).
- <Block-name> <block-dereference> - to access a block declared in the body of a <block-type-declaration>.
- <Block-name> - to access any other block.

Constraints:

A <subscript> must be present in a <table-item> or <block-table-item> if the type of the table is dimensioned.

A <subscript> in a <table-item>, <table-entry>, <block-table-item>, or <block-table-entry> must contain the same number of <indices> as there are <dimensions> in the <dimension-list> of the declaration of the table's type.

Furthermore, the type of each <index> must be equivalent to, or be implicitlyy: convertible to, the type of the corresponding <dimension> and the value of each index must be within the bounds specified for that dimension. If the designated table is a formal parameter and the <dimensions> were specified as *, the indices must be <integer-formulas> (even if bounds of an actual parameter on a particular invocation are of status type), and the value of each index must be in the range 0 through NN-1, where NN is the number of elements in that dimension of the actual parameter.

If the <table-item-name> in a <table-item> was declared in a <table-type-declaration>, the <table-item> must contain a <table-dereference> whose pointer is of the appropriate type.

If the <item-name> in a <block-item> was declared in a <block-type-declaration>, the <block-item> must contain a <block-dereference> whose pointer is of the appropriate type.

A reference to a <table-item> or <table-entry> must not access storage outside the bounds of the table containing that <table-item> or <table-entry>.

If the <table-name> in a <block-table> or <block-table-entry> was declared in a <block-type-declaration>, the <block-table> or <block-table-entry> must contain a <block-dereference> whose pointer is of the appropriate type.

If the <table-item-name> in a <block-table-item> was declared in a <block-type-declaration>, the <block-table-item> must contain a <block-dereference> whose pointer is of the appropriate type.

<Fbit> and <nbit> must not designate a substring beyond the bounds of the <bit-variable>. <Nbit> must be greater than zero.

<Fbyte> and <nbyte> must not designate a substring beyond the bounds of the <character-variable>. <Nbyte> must be greater than zero.

A <function-name> can be used as a <variable> only within the body of a function having that <function-name>, and then only as the left-hand side of an assignment statement. The other valid uses of <function-name> are described in Section 3.2.

A pointer to an undimensioned parallel or tight table type cannot be used in a <dereference>.

The value of a pointer used in a <dereference> must be in the implementation-defined set of valid values for pointers of its type. A pointer whose value is NULL cannot be dereferenced.

6.2 NAMED CONSTANTS

Syntax:

<named-constant> ::= <constant-item-name> (2.1.3)

| <constant-table-name> (2.1.3)

| <constant-table-item-name> [<subscript>] (6.1)

| <constant-table-name> (2.1.3) <subscript> (6.1)

| <control-letter> (4.2)

<constant-table-item-name> ::= <table-item-name> (2.1.2.3)

Semantics:

A <named-constant> designates a constant data object whose value can be used in a formula but cannot be changed.

A <constant-item-name> designates an object declared in a constant item declaration.

A <constant-table-name> designates an object declared in a constant table declaration.

A <constant-table-item-name> designates an item component of a constant table. If the table is dimensioned, the <subscript> indicates from which entry the item is to be obtained.

A <constant-table-name> followed by a <subscript> designates an entry in a dimensioned constant table.

A <control-letter> designates an object created in a <for-clause> whose <control-item> is a single letter.

Constraints:

A <subscript> must follow a <constant-table-item-name> if the table is dimensioned.

A <subscript> following a <constant-table-item-name> or <constant-table-name> must contain the same number of <indices> as there are <dimensions> in the <dimension-list> in the declaration of the table. Furthermore, the type of each <index> must be the same as the type of the corresponding <dimension> and the value of each <index> must be within the bounds specified for that <dimension>.

Constant tables and items selected from constant tables via subscripts cannot be used as compile-time values.

A <control-letter> may be referenced only within the <loop-statement> whose <for-clause> created that <control-letter>.

6.3 FUNCTION CALLS

Syntax:

<function-call>	::=	<user-defined-function-call>	
	¦	<intrinsic-function-call>	
	¦	<machine-specific-function-call>	
<user-defined-function-call>	::=	<function-name>	(3.2)
		[<actual-parameter-list>]	(4.5)
<intrinsic-function-call>	::=	<loc-function>	(6.3.1)
	¦	<next-function>	(6.3.2)
	¦	<bit-function>	(6.3.3)
	¦	<byte-function>	(6.3.4)
	¦	<shift-function>	(6.3.5)
	¦	<abs-function>	(6.3.6)
	¦	<sign-function>	(6.3.7)
	¦	<size-function>	(6.3.8)
	¦	<bounds-function>	(6.3.9)
	¦	<nwdsen-function>	(6.3.10)
	¦	<status-inverse-function>	(6.3.11)
	¦	<nent-function>	(6.3.12)
<machine-specific-function-call>	::=	<function-name>	(3.2)
		[<actual-parameter-list>]	(4.5)

Semantics:

Execution of a <function-call> causes invocation of a function. Any actual parameters are bound to the corresponding formal parameters as described in Section 3.3.

A <user-defined-function-call> causes invocation of a function defined in a <function-definition>. The type of the value returned by the function is the type specified by the <item-type-description> in the <function-heading> of the <function-definition>.

An <intrinsic-function-call> causes invocation of a language-defined function. A description of the language-defined functions is contained in the following sections. The type of the value returned by each function is described in the corresponding section.

A <machine-specific-function-call> causes invocation of a machine-specific function (see Section 3.5).

Constraints:

Actual parameters in the <function-call> must match the formal parameters of the called function in number, type, and parameter list position according to the rules given in Section 3.3.

6.3.1 LOC FUNCTION

Syntax:

<loc-function>	::=	LOC (<loc-argument>)	
<loc-argument>	::=	<named-variable>	(6.1)
	|	<statement-name>	(4.0)
	|	<procedure-name>	(3.1)
	|	<function-name>	(3.2)
	|	<block-reference>	(6.1)

Semantics:

The LOC function can be applied to the <loc-argument> to obtain the machine address of the word in which the <loc-argument> is stored. The type of the value returned by the LOC function for each <loc-argument> listed below is a typed pointer (e.g., type P TT if the <type-name> involved is TT):

- An <item> or <table-item> whose declared description is <type-name>.
- A <table> whose declaration includes <type-name> but excludes both <dimension-list> and <entry-specifier>.
- A <table-entry> of a data table or table type whose declaration includes <dimension-list> followed by <type-name> but excludes <entry-specifier>.
- A <dereference>.

Otherwise, the type of the value returned by the LOC function is P (i.e., an untyped pointer).

If the <loc-argument> is a <statement-name>, <procedure-name>, or <function-name> the <loc-function> yields an untyped pointer whose value is the machine address used to access the designated statement or subroutine.

Constraints:

The LOC of a subroutine whose name appears in an <inline-declaration>, or of a <statement-name> whose definition appears in such a subroutine, is implementation-defined.

Note:

The LOC function cannot be applied to an intrinsic function.

6.3.2 NEXT FUNCTION

Syntax:

<next-function>	::=	NEXT (<next-argument> , <increment-amount>)	
<next-argument>	::=	<pointer-formula>	(5.5)
	|	<status-formula>	(5.4)
<increment-amount>	::=	<integer-formula>	(5.1.1)

Semantics:

If the <next-argument> is a <pointer-formula>, the value returned by the NEXT function is the arithmetic sum of the representation of the <pointer-formula> plus the <increment-amount> * LOCSINWORD (i.e., the <pointer-formula> is treated as an integer). The type of the value returned is a pointer of the same type as the <next-argument>.

If the <next-argument> is a <status-formula> and the value of the <increment-amount> is N, the value returned by the NEXT function is the Nth successor (or predecessor if N is negative) of the value of the <status-formula> in this <status-list>. The type of the value is the same as the type of the <next-argument>.

Constraints:

The <next-argument> cannot be a <status-constant> that belongs to more than one status type (unless explicitly disambiguated with a <status-conversion>), nor can it be the <pointer-literal> NULL.

The type of the <status-formula> must be a status type with a default representation.

When the <next-argument> is a <status-formula>, the <increment-amount> must not cause the NEXT function to return a value out of range of the type of the <next-argument>.

The value of the <pointer-formula> and the value of the pointer result must be in the implementation-defined set of valid values for pointers of its type.

Note:

The value of the <increment-amount> may be negative.

Note that, assuming table tt is neither tight nor non-parallel

NEXT (LOC(tt(nn)), NWDSEN(tt)) = LOC(tt(nn+1))

6.3.3 BIT FUNCTION

Syntax:

<bit-function>	::=	BIT (<bit-formula> , <fbit> , <nbit>)	(5.2)
<fbit>	::=	<integer-formula>	(5.1.1)
<nbit>	::=	<integer-formula>	(5.1.1)

Semantics:

The BIT function selects a designated substring from the <bit-formula>. <Fbit> indicates the starting bit and <nbit> indicates the size of the substring. Bits are numbered from the left beginning with zero. The type of the value returned is a bit string of the same size as the <bit-formula>. The designated substring is right justified in the result and padded on the left with zero bits as necessary to fill the size.

Constraints:

<Fbit> and <nbit> must not designate a substring beyond the bounds of the <bit-formula>. <Nbit> must be greater than zero.

6.3.4 BYTE FUNCTION

Syntax:

<byte-function>	::=	BYTE (<character-formula> , <fbyte> , <nbyte>)	(5.3)
<fbyte>	::=	<integer-formula>	(5.1.1)
<nbyte>	::=	<integer-formula>	(5.1.1)

Semantics:

The BYTE function selects a designated substring from the <character-formula>. <Fbyte> indicates the starting character and <nbyte> indicates the size of the substring. Characters are numbered from the left beginning with zero. The type of the value returned is a character string of the same size as the <character-formula>. The designated substring is left justified in the result, and padded on the right with blanks as necessary to fill the size.

Constraints:

<Fbyte> and <nbyte> must not designate a substring beyond the bounds of the <character-formula> . <Nbyte> must be greater than zero.

6.3.5 SHIFT FUNCTIONS

Syntax:

<shift-function>	::=	<shift-direction> (<bit-formula> , <shift-count>)	(5.2)
<shift-direction>	::=	SHIFTL	
	|	SHIFTR	
<shift-count>	::=	<integer-formula>	(5.1.1)

Semantics:

The SHIFTL function performs a logical left shift of the <bit-formula> by the number of positions indicated by <shift-count>. The SHIFTR function performs a logical right shift of the <bit-formula> by the number of positions indicated by <shift-count>. In both cases, vacated bits are filled with zeros and bits shifted out are lost. If the <shift-count> is greater than or equal to the size of the <bit-formula>, the result is a bit string with all zero bits. The type of the value returned by a <shift-function> is the same as the type of the <bit-formula>.

Constraints:

The value of <shift-count> must be non-negative and less than or equal to MAXBITS.

6.3.6 ABS FUNCTION

Syntax:

<abs-function> ::= ABS (<numeric-formula>) (5.1)

Semantics:

The ABS function produces a value that is the absolute value of the <numeric-formula>. The result is equivalent to - <numeric-formula> if <numeric-formula> is negative and equivalent to + <numeric-formula> otherwise.

6.3.7 SIGN FUNCTION

Syntax:

<sign-function> ::= SGN (<numeric-formula>) (5.1)

Semantics:

The SGN function returns a value according to the following rules:

Numeric Formula	Value
> 0	+1
= 0	0
< 0	-1

The type of the value is S 1.

6.3.8 SIZE FUNCTIONS

Syntax:

<size-function> ::= <size-type>
(<size-argument>)

<size-type> ::= BITSIZE

| BYTESIZE

| WORDSIZE

<size-argument>	::=	<formula>	(5.0)
	|	<block-name>	(2.1.4)
	|	<type-name>	(2.1.1.7)

Semantics:

The BITSIZE, BYTESIZE and WORDSIZE functions return the logical size of the <size-argument> in bits, bytes, and words respectively. The type of the value returned is S MAXINTSIZE. The logical BITSIZE of each data type in the language will be described below. The logical BYTESIZE is equal to BITSIZE/BITSINBYTE if BITSIZE MOD BITSINBYTE = 0 and BITSIZE/BITSINBYTE+1 otherwise. Similarly, the logical WORDSIZE is equal to BITSIZE/BITSINWORD if BITSIZE MOD BITSINWORD = 0 and BITSIZE/BITSINWORD+1 otherwise.

Bit: The BITSIZE of an object of type B NN is NN

Integer: The BITSIZE of an object of type U NN is NN and S NN is NN+1

Fixed: The BITSIZE of an object of type A MM, NN is MM+NN+1

Float: The BITSIZE of a float object is the number of bits of storage the object actually occupies.

Character: The BITSIZE of an object of type C NN is NN*BITSINBYTE.

Pointer: The BITSIZE of a pointer object is BITSINPOINTER.

Status: The BITSIZE of a status object is the minimum number of bits to be allocated, as defined in Section 2.1.1.6, for <status-formulas> using unsigned integer representation, that minimum plus one when signed integer representation is required.

Table: The BITSIZE of a table or table entry that is not tightly structured is the number of bits from the leftmost bit of the first word occupied by the table or table entry to the rightmost bit of the last word occupied by the table or table entry. The BITSIZE of a tightly structured table entry is <bits-per-entry>. The BITSIZE of a tightly structured table is the number of bits from the leftmost bit of the first word occupied by the table to the rightmost bit of the last entry, where the last entry occupies <bits-per-entry> bits. Note: the BITSIZE of a <table-entry> whose type class is not table is the BITSIZE of the item specified by the underlying <item-type-description>.

Block: The BITSIZE of a block is NN * BITSINWORD, where NN is the number of words the block occupies.

Constraints:

A BITSIZE function must not be applied to a <size-argument> whose size in words exceeds MAXINT(MAXINTSIZE)/BITSINWORD.

A BYTESIZE function must not be applied to a <size-argument> whose size in words exceeds MAXINT(MAXINTSIZE)/BYTESINWORD.

6.3.9 BOUNDS FUNCTIONS

Syntax:

```
<bounds-function>       ::= <which-bound>
                            ( <bounds-argument> ,
                            <dimension-number> )

<which-bound>           ::= LBOUND

                          | UBOUND

<bounds-argument>       ::= <table-name>                          (2.1.2)

                          | <table-type-name>                     (2.2)

<dimension-number>      ::= <compile-time-integer-formula>        (5.1.1)
```

Semantics:

The LBOUND function returns the lower bound of the specified dimension of the designated table. The UBOUND function returns the upper bound of the specified dimension of the designated table. A <dimension-number> of zero refers to the leftmost <dimension> in that table's <dimension-list>; a <dimension-number> of one designates the next-to-leftmost <dimension> in the list, etc. The type of the returned value is the type of the designated dimension. If the table is a formal parameter with a * dimension, LBOUND will always return zero, and UBOUND will return NN-1, where NN is the number of elements in that dimension of the actual parameter.

Constraints:

The <dimension-number> must be greater than or equal to 0 and less than the number of dimensions in the designated table.

6.3.10 NWDSEN FUNCTION

Syntax:

```
<nwdsen-function>       ::= NWDSEN ( <nwdsen-argument> )
```

```
<nwdsen-argument>      ::=  <table-name>                    (2.1.2)
                          |  <table-type-name>               (2.2)
```

Semantics:

The NWDSEN function returns the number of words of storage allocated to each entry in the named table or table type. The return type is S with default size.

6.3.11 STATUS INVERSE FUNCTIONS

Syntax:

```
<status-inverse-function>  ::=  FIRST (
                                <status-inverse-argument> )

                           |    LAST (
                                <status-inverse-argument> )

<status-inverse-argument>  ::=  <status-formula>          (5.4)

                           |    <status-type-name>        (2.1.1.6)
```

Semantics:

The FIRST function gives the value of the lowest-valued <status-constant> in the <status-list> associated with the <status-inverse-argument>. The LAST function gives the value of the highest-valued <status-constant> in the <status-list> associated with the <status-inverse-argument>.

The return value has the type indicated by the <status-inverse-argument>.

6.3.12 NENT FUNCTION

Syntax:

```
<nent-function>        ::=  NENT ( <nent-argument> )

<nent-argument>        ::=  <table-name>                  (2.1.2)
                          |  <table-type-name>            (2.2)
```

Semantics:

The <nent-function> returns the number of entries in the designated table. For multi-dimensioned tables, the value returned is the product of the numbers of entries of the individual dimensions. For tables with * dimensions the type of the value returned is S IMPLINTSIZE(MINSIZE(MAXTABLESIZE)). In all other

cases the type is S IMPLINTSIZE(MINSIZE(nent)), where nent is the number of entries in the table.

7.0 TYPE MATCHING AND TYPE CONVERSIONS

Syntax

```
<bit-conversion>        ::= <bit-type-conversion>
                          | <rep-conversion>
<bit-type-conversion>   ::= (* <bit-type-description> *)      (2.1.1.4)
                          | <bit-type-name>                   (2.1.1.4)
                          | B
<integer-conversion>    ::= (* <integer-type-description> *)  (2.1.1.1)
                          | <integer-type-name>               (2.1.1.1)
                          | S
                          | U
<floating-conversion>   ::= (* <floating-type-description> *) (2.1.1.2)
                          | <floating-type-name>              (2.1.1.2)
                          | F
<fixed-conversion>      ::= (* <fixed-type-description> *)    (2.1.1.3)
                          | <fixed-type-name>                 (2.1.1.3)
<character-conversion>  ::= (* <character-type-description> *)(2.1.1.5)
                          | <character-type-name>             (2.1.1.5)
                          | C
<status-conversion>     ::= (* <status-type-name> *)          (2.1.1.6)
                          | <status-type-name>                (2.1.1.6)
<pointer-conversion>    ::= (* <pointer-type-description> *)  (2.1.1.7)
                          | <pointer-type-name>               (2.1.1.7)
                          | P
<table-conversion>      ::= (* <table-type-name> *)           (2.1.2)
                          | <table-type-name>                 (2.1.2)
```

<rep-conversion> ::= REP

Semantics:

In Section 2.1, the definition of type was given. In some cases, implicit conversions will be performed to achieve type equivalence. In this section, for each type class, rules will be given regarding when two types are the same, when an object of one type will be implicitly converted to another type, and when and how an object of one type can be explicitly converted to another type. Implicit conversions will never be performed on arguments to explicit conversions or when the types of the data objects are required to match exactly. With all the conversions (both implicit and explicit), if the value produced after conversion is not in the range of values of the type being converted to, the conversion is illegal.

For purposes of type equivalence, a user-defined <type-name> is considered an abbreviation for its specification.

A <formula> may be explicitly converted to another type by enclosing it in parentheses and preceding it with appropriate conversion. Note that if the conversion does not consist of a single letter or name, it must be enclosed in (* and *).

Omitted attribute specifiers in type conversions imply the same default values as for declarations of those types.

Type equivalence and conversion rules for each of the J73 type classes are as follows:

Bit (B)

Type Equivalence: Two bit types are equivalent if their size attributes are equal.

Implicit Conversions: A bit string will be implicitly converted to a bit string with a different size attribute, with truncation on the left or padding with zeros on the left. Implicit truncation is not permitted when the syntax requires a <boolean-formula>.

Explicit Conversions: Any data object except a block may be explicitly converted to a bit string with a <bit-conversion>. A <bit-conversion> may be either a <bit-type-conversion> or a <rep-conversion>.

A <bit-type-conversion> to a type B NN takes the rightmost NN bits of the data object's representation. If there are fewer than NN bits, the object will be padded on the left with zeroes. The default value for NN is 1. A <bit-type-conversion> may be applied to a data object of any type. If the object being converted is a table or table entry, all "filler" bits (i.e., bits

that contribute to the size of the table but that are not part of the component objects' sizes as declared) are included in the string. If the object to be converted is of type class character, filler bits between bytes and unused bytes following the end of the string are not included.

A <rep-conversion> provides a means of obtaining the representation of a data object. A <rep-conversion> treats a data object as a bit string whose size is the number of bits actually occupied by the object. This includes all filler bits and the bits in the unused (but allocated) bytes following the ends of character strings. For all objects whose type class is table, the number of bits in the bit string is the same as the BITSIZE OF THE OBJECT. For all <table-entries> whose type class is not table, the number of bits in the bit string is the total number of bits (including filler bits) in the table entry. A <rep-conversion> can appear in the target of an assignment statement (see Section 6.1). A <rep-conversion> can be applied to <named-variables> only; further, it cannot be applied to tables declared with * dimensions, to entries in parallel tables, or to tables whose size in bits exceeds MAXBITS.

Integer (S and U)

Type Equivalence: Two integer types are equivalent if they are both S or U and if their size attributes are equal.

Implicit Conversions: An integer type will be implicitly converted to any other integer type.

Explicit Conversions: An <integer-conversion> is used to explicitly convert a data object to an integer type. The conversion can be applied to <real-literals> and to objects of bit, integer, fixed, float, and pointer only.

A bit string will be treated as representing the value of the integer type if the size of the bit string is less than or equal to the BITSIZE of the integer type. Otherwise, the conversion is illegal. If the size of the bit string is less than the BITSIZE of the integer type, the bitstring will be padded on the left with zeroes.

An integer, fixed, or floating data object will be converted to the integer type, with truncation or rounding if specified.

Converting a pointer to an integer type is equivalent to first converting the pointer to type B BITSINPOINTER and then converting the bit string to integer.

Floating (F)

Type Equivalence: Two floating types are equivalent if their precision attributes are equal.

Implicit Conversions: A floating type will be implicitly converted to a floating type of the same or greater precision regardless of the round-or-truncate attribute. A <real-literal> will be implicitly treated as a <floating-literal> in the contexts specified in Section 8.3.1. (Implicit floating conversions do not change numeric values although they may cause a change in how the value is represented.)

Explicit Conversions: A <floating-conversion> is used to explicitly convert a data object to a floating data type. The conversion can be applied to <real-literals> and to objects of bit, integer, fixed, and float types only.

A bit string will be treated as representing the value of the floating type if the size of the bit string equals the BITSIZE of the floating type. Otherwise the conversion is illegal.

An integer, fixed, or floating data object will be converted to the floating type, with truncation or rounding as specified in the <floating-conversion>. Rounding and truncation are performed with respect to the implemented precision of the type specified by the <floating-conversion>.

Fixed (A)

Type Equivalence: Two fixed point types are equivalent if their scale attributes are equal and their fraction attributes are equal.

Implicit Conversions: A fixed point type will be implicitly converted to another fixed point type if the scale and fraction attributes of the target type are both at least as large as those of the source type. A <real-literal> will be implicitly treated as a <fixed-literal> in the contexts specified in Section 8.3.1. Implicit fixed conversions do not change the numeric value represented except when the implemented precision of the result value is less than the implemented precision of the value being converted (see Section 2.1.1.3); in this case, rounding or truncation occurs with respect to the implemented precision of the converted value. This situation occurs only when assigning to a packed fixed table item (in an assignment statement, loop <control-variable>, table preset, or output parameter); the <round-or-truncate>

attribute of the table item determines whether the assigned value is rounded or truncated.

Explicit Conversions: A <fixed-conversion> is used to explicitly convert a data object to a fixed point data type. The conversion can be applied to <real-literal> and to objects of bit, integer, fixed, and float types only.

A bit string will be treated as representing the value of the specified fixed point type if the size of the bit string equals the BITSIZE of the fixed point type. Otherwise, the conversion is illegal.

An integer, fixed, or floating data object will be converted to the specified fixed point type, with truncation or rounding as specified in the <fixed-conversion>. Rounding and truncation are performed with respect to the implemented precision of the type specified by the <fixed-conversion>.

Character (C)

Type Equivalence: Two character types are equivalent if their size attributes are equal.

Implicit Conversions: A character string will be implicitly converted to a string with a different size attribute, with truncation on the right or padding with blanks on the right.

Explicit Conversions: A <character-conversion> is used to explicitly convert a data object to a character data type. The conversion can be applied to objects of type bit or character only.

A bit string will be treated as representing the value (excluding filler bits between bytes) of the character type if the size of the bit string equals the BITSIZE of the character type. Otherwise the conversion is illegal.

A character string will be converted to type C NN by taking the leftmost NN characters. If there are fewer than NN characters, the value is padded on the right with blanks.

Pointer (P)

Type Equivalence: Two pointer types are equivalent if they are both untyped pointers or if they are both typed pointers referring to the same <type-declaration>.

Implicit Conversions: A typed pointer will be implicitly converted to an untyped pointer.

Explicit Conversions: A <pointer-conversion> is used to explicitly convert a data object to a pointer type. The conversion can be applied to bit, integer, or pointer data objects only.

A bit string will be treated as representing the value of the pointer type if the size of the bit string equals the BITSIZE of the pointer type. Otherwise the conversion is illegal.

Converting an integer to a pointer is equivalent to first converting the integer to type B BITSINPOINTER and then converting the bit string to a pointer.

Converting a pointer to a different pointer type means that the pointer will be considered as a pointer of the specified type.

Status

Type Equivalence: Two status types are equivalent if (1) they both have default representation, their size attributes are the same, and both <status-lists> contain the same <status-constants> in the same order, or (2) they both have identical programmer-specified representations, their size attributes are the same, and both <status-lists> contain the same <status-constants>.

Implicit Conversions: A status type will be implicitly converted to a status type that differs only in its size attribute. Furthermore, a status constant belonging to more than one status type is implicitly disambiguated in the following contexts: (1) when it is the source value of an assignment statement, it takes the type of the target variable; (2) when it is an actual parameter, it takes the type of the corresponding formal parameter; (3) when it is in a table <subscript> or <preset-index-specifier>, it takes the type of the corresponding <dimension> in that table's declaration; (4) when it is a loop <initial-value>, it takes the type of the <control-variable>; (5) when it is in an <item-preset> or <table-preset>, it takes the type of the item or table item being initialized; (6) when it is an operand of a <relational-operator>, it takes the type of the other operand if that type is uniquely determined independent of the former operand; (7) when it is in a <case-index-group>, it takes the type of the <case-selector-formula>; and (8) when it is a <lower-bound> or <upper-bound>, it takes the type of the other bound if that type is uniquely determined independent of the former bound.

Explicit Conversions: A <status-conversion> is used to explicitly convert a data object to a status type. The conversion can be applied to bit or status data objects only.

A bit string will be treated as representing the representational value of the status type if the size of the bit string equals the BITSIZE of the status type and the value of the bit string is within the range of values of the status type. Otherwise the conversion is illegal.

A <status-conversion> may be used to assert the type of a status object. This will be required when a status constant belongs to more than one type and it is used in a context other than those enumerated above under implicit conversions. Except for status objects whose types differ only in their size attributes, a status object cannot be converted to a different status type without first converting it to a bit string.

Table

Type Equivalence: Two tables have equivalent types if they are both ordinary or both specified, their <structure-specifier> attribute is the same, they have the same number of dimensions, they have the same number of elements in each dimension, they have the same number of items in the same textual order in each entry, the types (including attributes) of the items are equivalent, the (explicit or implied) packing specifier on each of the items is the same (for ordinary tables), the !ORDER directive is either present in both tables or absent in both tables, the <words-per-entry> attribute is the same (for specified tables), and the location-specifiers of the items are the same (for specified tables). (Note that the names of the items, as well as the types and bounds of the dimensions, need not be the same.) A table entry is considered to have no dimensions. A table whose entry contains an item-declaration is not considered equivalent in type to a table whose entry is declared using an unnamed item description.

A <dimension> of * in a formal-parameter table has the same number of elements as the corresponding <dimension> (which may be a <dimension> of *) in the corresponding actual-parameter table.

Implicit Conversions: No implicit conversions are performed.

Explicit Conversions: A bit or table data object may be explicitly converted to a table type with a <table-conversion>.

A bit string will be treated as representing the value of the table type if the size of the bit string equals the BITSIZE of the table type. Otherwise the conversion is illegal.

A <table-conversion> may be applied to a table object of that type merely to assert its type. (A table object cannot be converted to a different table type without first converting it to a bit string).

8.0 BASIC ELEMENTS

8.1 CHARACTERS

Syntax:

```
<character>     ::= <letter>
                  | <digit>
                  | <mark>
                  | <other-character>
<letter>        ::= A | B | C | D | E | F
                  | G | H | I | J | K | L
                  | M | N | O | P | Q | R
                  | S | T | U | V | W | X
                  | Y | Z
<digit>         ::= 0 | 1 | 2 | 3 | 4 | 5
                  | 6 | 7 | 8 | 9
<mark>          ::= + | - | * | / | > | <
                  | = | @ | . | : | , | ;
                  | ( | ) | ' | " | % | !
                  | $ | blank
```

Semantics:

The text of a J73 <complete-program> is a continuous stream of <characters>. However, in some contexts, the end of an input record has significance (see Section 8.2).

Note that in the standard character set for the language <letters> are defined to be upper case letters only. <Marks> are used either alone or in conjunction with other characters as operators, delimiters, and separators. <Other-characters> are the remaining implementation-dependent characters, which are accepted within <character-literals> and <comments>, and which may also be used as described below. Each implementation must define these characters, as well as the ordering of all <characters> in a collating sequence.

Some of the standard characters are not universally available; therefore, the following standard alternates are defined:

Standard Character	Alternates
@	^ or ?
'	-> or _
"	≠
!	V
%	¬
:	%

If any of the above standard characters are unavailable on a particular machine, one of the recommended alternates for that character must be used. (The first column of alternates is intended for the CDC standard 63 and 64 character sets; the alternates ? and _ are intended for the Univac 1108.) If the : is replaced, the % must also be replaced.

An implementation that has both lower and upper case letters available may permit use of lower case letters outside <character-literals> and <comments> provided that the implementation always considers them interchangeable with their corresponding upper case letters (e.g., XX, Xx, xX, and xx denote the same name). In all <character-literals> and <comments>, including those constructed via <define-calls>, lower case letters must be treated as distinct <other-characters>.

An implementation that has square brackets available may allow [to be used for (* and] to be used for *) but may not prohibit the use of the (* and *).

Constraints:

If a left bracket is substituted for (*, then a right bracket must be substituted for the corresponding *). If a right bracket is substituted for *), then a left bracket must be substituted for the corresponding (*.

8.2 SYMBOLS

Syntax:

<symbol>	::=	<name>	(8.2.1)
	|	<reserved-word>	(8.2.2)
	|	<operator>	(8.2.3)
	|	<literal>	(8.3)
	|	<status-constant>	(2.1.1.6)

¦	<comment>	(8.4)
¦	<define-string>	(2.4)
¦	<define-call>	(2.4.1)
¦	<letter>	(8.1)
¦	<separator>	(8.2.4)

Semantics:

<Characters> are combined into <symbols> to form the vocabulary of the language. <Symbols> are indivisible units and cannot contain blanks, except as noted in Section 8.5. Only <comments>, <define-strings>, define parameters enclosed in quotation marks, <bit-literals>, and <character-literals> may extend across multiple input records; all other symbols are terminated by the end of an input record.

8.2.1 NAMES

Syntax:

<name>	::=	<letter-or-$> <letter-digit-$-or-prime>...	
<letter-or-$>	::=	<letter>	
	¦	$	
<letter-digit-$-or-prime>	::=	<letter>	(8.1)
	¦	<digit>	(8.1)
	¦	$	
	¦	'	

Semantics:

<Names> are words having programmer-supplied spellings. <Names> are used to denote entities in the <complete-program>.

Only the first 31 characters of a J73 <name> are used to determine uniqueness. Additional characters are permitted, but are ignored.

For external names, an implementation may further restrict the number of initial characters that determine uniqueness.

A dollar sign in a <name> is translated to an implementation-dependent representation. This translation of the dollar sign permits the use of a

character in a <name> that might otherwise be unrepresentable in the language. If, for example, external names in a given system were prefixed by the character '.' a J73 implementation on that system might choose to represent '$' when it occurs in a name by the representation for '.'. Thus, the name '$$ABC' occurring in a source program would be translated '..ABC'.

Constraints:

A <name> cannot have the same spelling as a <reserved-word> (see Section 8.2.2).

8.2.2 RESERVED WORDS

Syntax:

```
<reserved-word>          ::=  ABORT  |  ABS  |  AND  |  BEGIN  |  BIT
                            |  BITSIZE  |  BLOCK  |  BY  |  BYREF
                            |  BYRES  |  BYTE  |  BYTESIZE  |  BYVAL
                            |  CASE  |  COMPOOL  |  CONDITION*
                            |  CONSTANT  |  DEF  |  DEFAULT  |  DEFINE
                            |  ELSE  |  ENCAPSULATION*  |  END  |  EQV
                            |  EXIT  |  EXPORTS*  |  FALLTHRU  |  FALSE
                            |  FIRST  |  FOR  |  FREE*  |  GOTO
                            |  HANDLER*  |  IF  |  IN*  |  INLINE
                            |  INSTANCE  |  INTERRUPT*  |  ITEM
                            |  LABEL  |  LAST  |  LBOUND  |  LIKE
                            |  LOC  |  MOD  |  NENT  |  NEW*
                            |  NEXT  |  NOT  |  NULL  |
                            |  NWDSEN  |  OR  |  OVERLAY  |  PARALLEL
                            |  POS  |  PROC  |  PROGRAM  |  PROTECTED*
                            |  READONLY*  |  REC  |  REF  |  REGISTER*
                            |  RENT  |  REP  |  RETURN  |  SGN
                            |  SHIFTL  |  SHIFTR  |  SIGNAL*
```

| START | STATIC | STATUS | STOP

| TABLE | TERM | THEN | TO*

| TRUE | TYPE | UBOUND | UPDATE*

| WHILE | WITH* | WORDSIZE

| WRITEONLY* | XOR | ZONE*

Semantics:

<Reserved-words> have language-defined meanings and cannot be used as <names>.

Those reserved words followed by an * in the above list are reserved in order to maintain upward compatibility with future extensions to the language and currently have no meaning in J73.

8.2.3 OPERATORS

Syntax:

```
<operator>                    ::= <arithmetic-operator>
                                | <bit-operator>
                                | <relational-operator>
                                | <dereference-operator>
                                | <assignment-operator>
<arithmetic-operator>         ::= <plus-or-minus>
                                | <multiply-divide-or-mod>
                                | <multiply-or-divide>
                                | **
<plus-or-minus>               ::= + | -
<multiply-divide-or-mod>      ::= * | / | MOD
<multiply-or-divide>          ::= * | /
<bit-operator>                ::= <logical-operator>
                                | NOT
```

```
<logical-operator>                  ::= AND | OR | XOR | EQV

<relational-operator>               ::= <equal-or-not-equal-operator>
                                        | < | > | <= | >=

<equal-or-not-equal-operator>       ::= = | <>

<dereference-operator>              ::= @

<assignment-operator>               ::= =
```

Semantics:

The meanings of these operators are given in Sections 4, 5, and 6. The order of combination of operators and operands is determined by parentheses and by the operators' precedence. The operation implied by an operator at one precedence level is combined before the operation implied by an operator at a lower level. Within a particular precedence level, operations are combined from left to right if the !LEFTRIGHT directive is in effect and in an implementation-dependent order if the !REARRANGE directive is in effect.

Precedence of operators is defined by the syntax of the language and is summarized below:

6	@, subscripting, function calls
5	**
4	*, /, MOD
3	+, -
2	=, <>, <, >, <=, >=
1	NOT, AND, OR, EQV, XOR
0	assignment

8.2.4 SEPARATORS

Syntax:

```
<separator>                         ::= ( | ) | (* | *)
                                        | : | , | ; | !
```

Semantics:

<Separators> are used for the following purposes in J73:

( )	Expression grouping, list delimiters, status constants, position brackets, subscripts, case labels
(* *)	Type conversions
:	Statement name, case label, and preset index terminator; loop control separator; overlay, dimension, subrange, and parameter separator
,	List separator
;	Statement, declaration, and directive terminator
!	Directive indicator, formal define parameter marker

8.3 LITERALS

Syntax:

<literal>	::=	<numeric-literal>	(8.3.1)
	¦	<bit-literal>	(8.3.2)
	¦	<boolean-literal>	(8.3.3)
	¦	<character-literal>	(8.3.4)
	¦	<pointer-literal>	(8.3.5)

Semantics:

<Literals> are data objects whose value and type are inherent in the form of the <symbol> itself. Their values are known at compile time, and, like other compile-time values, cannot be altered during execution.

8.3.1 NUMERIC LITERALS

Syntax:

<numeric-literal>	::=	<integer-literal>
	¦	<floating-literal>
	¦	<fixed-literal>

```
<integer-literal>       ::= <number>

<number>                ::= <digit>...                          (8.1)

<floating-literal>      ::= <real-literal>

<real-literal>          ::= <digit>... <exponent>               (8.1)

                          | <fractional-form>
                            [<exponent>]

<exponent>              ::= E [<sign>] <number>

<sign>                  ::= + | -

<fractional-form>       ::= <digit>... .                        (8.1)

                          | [<digit>...] . <digit>...           (8.1)

<fixed-literal>         ::= <real-literal>
```

Semantics:

An <integer-literal>, LL, denotes a decimal value. Its type is S NN, where NN is IMPLINTSIZE(MINSIZE(LL)).

The type of a <real-literal> or a <real-literal> preceded by a <sign> is determined by the context in which the literal appears, namely:

- when the literal is used as a preset value, it is implicitly converted to the type of the object being preset;
- when the literal is used as an assignment value, it is implicitly converted to the type of the target being assigned a value;
- when the literal is an operand of an infix relational or numeric operator and the other operand is not a real-literal, it is converted to the type of the other operand;
- when the literal is an actual parameter, it is converted to the type of the formal parameter;
- when a literal is the <initial-value> of a loop <control-clause>, it is converted to the type of the <control-variable>;
- when the literal is the argument of an explicit integer, fixed, or floating conversion, it is converted to the specified type.

If the type of an optionally signed <real-literal> is not determined contextually, it is considered to be a floating type with default precision.

A <real-literal> denotes a decimal value. If an <exponent> is present, the decimal value preceding the <exponent> is multiplied by 10 to the value specified in the <exponent>.

For <real-literals>, non-<exponent> digits in excess of MAXSIGDIGITS will be treated as zeroes in computing the fixed or floating value to be represented.

Contextual determination of the type of a real-literal will not be affected by the presence or absence of the <rearrange-directive>.

Constraints:

<Real-literals> may be implicitly converted to fixed or floating values only.

The value of an <integer-literal> with size SS must not exceed MAXINT(SS).

The value of a <floating-literal> with precision PP must not exceed MAXFLOAT(PP).

The value of a <fixed-literal> with scale SS and fraction FF must not exceed MAXFIXED(SS,FF).

Examples:

```
ITEM FF F 24 = -0.1;            "equivalent to presetting
                                     with (*F 24*) (-0.1)"
ITEM RR F,R 24 = -0.1;          "-0.1 is rounded to a 24 bit
                                     mantissa"
ITEM TT F,T 24 = -0.1;          "-0.1 is truncated toward minus
                                     infinity"
CONSTANT ITEM CC F,R 24 = 2.5;
ITEM JJ F,R 24 = CC + .3;       ".3 is converted to CC's type"
IF RR > .3; ...                 ".3 is rounded to a 24 bit mantissa"
```

Note that if II is an integer item, then II = 2.5 is illegal, since a <real-literal> cannot be implicitly converted to an integer value.

8.3.2 BIT LITERALS

Syntax:

```
<bit-literal>     ::= <bead-size> B ' <bead>... '

<bead-size>       ::= 1 | 2 | 3 | 4 | 5

<bead>            ::= <digit>
                    | A | B | C | D | E | F
                    | G | H | I | J | K | L
```

| M | N | O | P | Q | R

| S | T | U | V

Semantics:

A <bit-literal> represents a bit string value. A <bit-literal> is composed of a string of <beads> whose <bead-size> in bits is indicated in the specification of the literal. The total size of the <bit-literal> is the <bead-size> times the number of beads enclosed within the primes.

The <beads> of a <bit-literal> can be specified as one to five bits in size. The <digit> preceding the B indicates the <bead-size>. Only those <beads> whose value will fit in the <bead-size> indicated are permitted. The digits 0 - 9 represent their actual values; the letters A - V represent the values 10 - 31 (see Table 8-1).

8.3.3 BOOLEAN LITERALS

Syntax:

<boolean-literal> ::= TRUE

| FALSE

Semantics:

<Boolean-literals> represent the two possible truth values. TRUE is equivalent to 1B'1', and FALSE is equivalent to 1B'0'.

8.3.4 CHARACTER LITERALS

Syntax:

<character-literal> ::= ' <character>... ' (8.1)

Semantics:

<Character-literals> denote strings of character values.

<Character-literals> can contain any <character> (including blank) that is representable in an implementation. A prime character (') is represented within a <character-literal> by two consecutive primes. The size of a <character-literal> in bytes is the number of characters represented within the containing primes (two consecutive primes represent one character). The encoding of characters is implementation-dependent.

TABLE 8-1. Bit-Literal Bead Values

Bead	Minimum Bead Size	Binary Value	Bead	Minimum Bead Size	Binary Value
0	1	0	G	5	10000
1	1	1	H	5	10001
2	2	10	I	5	10010
3	2	11	J	5	10011
4	3	100	K	5	10100
5	3	101	L	5	10101
6	3	110	M	5	10110
7	3	111	N	5	10111
8	4	1000	O	5	11000
9	4	1001	P	5	11001
A	4	1010	Q	5	11010
B	4	1011	R	5	11011
C	4	1100	S	5	11100
D	4	1101	T	5	11101
E	4	1110	U	5	11110
F	4	1111	V	5	11111

8.3.5 POINTER LITERALS

Syntax:

<pointer-literal> ::= NULL

Semantics:

Any pointer item, regardless of its attribute, can have the value NULL, which indicates that the item points to no object.

8.4 COMMENTS

Syntax:

<comment> ::= " [<character>...] " (8.1)

| % [<character>...] % (8.1)

Semantics:

A <comment> has no semantic effect.

A <comment> in a <define-string> or <actual-define-parameter> is interpreted as part of the character sequence to be substituted when the <define-call> is expanded.

A <comment> can appear before any <symbol>, subject to the constraints below.

Constraints:

A <comment> delimited by a quotation mark (") is not permitted between a <define-name> and a <define-string> in a <define-declaration>, or within the <actual-parameter-list> in a <define-call>.

A <comment> delimited by a quotation mark cannot contain a quotation mark, and a <comment> delimited by a percent character (%) cannot contain a percent character.

8.5 BLANKS

One or more blanks can be placed between <symbols>. Blanks occurring between <symbols> have no semantic meaning.

Constraints:

Blanks cannot appear within <symbols> except in <character-literals>, <define-strings>, <define-calls>, and <comments>.

One or more blanks must appear between any two <symbols> if the absence of blanks could cause them to be interpreted as a single legal <symbol>, except that whether (* represents one or two <symbols> is contextually determined, e.g., (* represents two symbols in the following contexts:

```
TABLE  AA (*) ...;
ITEM ... POS (*, 0);
```

9.0 DIRECTIVES

Syntax:

<directive>	::=	<compool-directive>	(9.1)
	¦	<copy-directive>	(9.2.1)
	¦	<skip-directive>	(9.2.2)
	¦	<begin-directive>	(9.2.2)
	¦	<end-directive>	(9.2.2)
	¦	<linkage-directive>	(9.3)
	¦	<trace-directive>	(9.4)
	¦	<interference-directive>	(9.5)
	¦	<reducible-directive>	(9.6)
	¦	<nolist-directive>	(9.7.1)
	¦	<list-directive>	(9.7.1)
	¦	<eject-directive>	(9.7.1)
	¦	<listinv-directive>	(9.7.2)
	¦	<listexp-directive>	(9.7.2)
	¦	<listboth-directive>	(9.7.2)
	¦	<base-directive>	(9.8)
	¦	<isbase-directive>	(9.8)
	¦	<drop-directive>	(9.8)
	¦	<leftright-directive>	(9.9)
	¦	<rearrange-directive>	(9.9)
	¦	<initialize-directive>	(9.10)
	¦	<order-directive>	(9.11)

Semantics:

<Directives> are used to provide supplemental information to a compiler about the <complete-program>, and to provide compiler control.

Each implementation can specify <directives> in addition to those described here, but each must conform to the general form for a <directive>. <Directives> begin with an exclamation point and terminate with a semicolon, and the word following the exclamation point must not duplicate that of any language-defined directive.

<Directives> are permitted to appear in J73 programs in the following places: immediately after BEGIN, END, START, DEF, REF, ELSE, FALLTHRU, and semicolon symbols, and in <case-statements> immediately following a <case-index-group> or the first colon in a <default-option>. (None of these uses are indicated in the syntax equations.) Note, however, that certain specific directives place further restrictions on where they can appear.

Like <define-calls>, <directives> each perform their function but have no other effect on syntactic analysis. For example, in the sequence "START <copy-directive> <compool-directive>", validity of the visible <compool-directive> can only be determined after the substitution specified by the <copy-directive>.

Directives affect program source from their occurrence to either end of module or occurrence of a superseding directive, unless different limits are explicitly specified for the particular type of directive.

9.1 COMPOOL DIRECTIVES

Syntax:

```
<compool-directive>       ::= !COMPOOL
                              [<compool-directive-list>]  ;

<compool-directive-list>  ::= [<compool-file-name>]
                              <compool-declared-name>,...

                            | (  [<compool-file-name>]  )

<compool-declared-name>   ::= <name>                          (8.2.1)

                            | (  <name>  )                    (8.2.1)

<compool-file-name>       ::= <character-literal>             (8.3.4)
```

Semantics:

A <compool-directive> is used to access definitions in a compool module.

If the <compool-directive-list> is omitted, all <names> in an implicit unnamed compool are to be made available.

A <compool-file-name> is an implementation-dependent file name that specifies the desired compool. If it is omitted, an implicit unnamed compool is assumed. A <compool-file-name> enclosed in parentheses implies that all <names> in the compool are to be made available. (This does not include <names> used in the compool that were obtained from other compools.)

If the <compool-directive> contains a list of <compool-declared-names>, only those names (except as noted below) will be made available.

If a <compool-declared-name> is the name of an item, table, or block declared with a <type-name>, that <type-name> is also made available if it is declared in that compool. (For pointer items, this includes the name of the pointed-to-type). If a <compool-declared-name> is a <table-item-name>, the name of the table in which it is contained is also made available. If a table name is made available, any <status-lists> and <status-type-names> associated with its <dimensions> are also made available, provided they are declared in the designated compool.

If a <compool-declared-name> is the name of a table or block and is parenthesized, all names declared in the table or block will be made available, as well as all type names referenced in the table or block, provided they are declared in the designated compool. If a <compool-declared-name> is a <table-type-name> or <block-type-name>, the names of these components will be made available whether or not the name is parenthesized.

If a status item name is made available, its associated <status-list> and <status-type-name> (if any) will also be made available, if they were declared in the designated compool.

If a <compool-declared-name> is the name of a subroutine, any <type-names> associated with that subroutine's formal parameters and return value will also be made available, if they are declared in the designated compool.

Constraints:

A <compool-directive> must only occur immediately after START or immediately following another <compool-directive>.

The <compool-declared-names> must have been declared in the designated compool.

A <compool-declared-name> cannot be the name of a component declared in a <type-declaration>, nor can it be the name of a formal parameter of a subroutine.

9.2 TEXT DIRECTIVES

9.2.1 COPY DIRECTIVES

Syntax:

```
<copy-directive>          ::= !COPY
                              <character-literal>   ;     (8.3.4)
```

Semantics:

The <copy-directive> is used to copy the contents of a text file into a program. The <copy-directive> can be viewed as a <define-call>; it is expanded

at the point of its occurrence by substituting the entirety of the file being copied. The <character-literal> is an implementation-dependent file name.

9.2.2 SKIP, BEGIN, AND END DIRECTIVES

Syntax:

```
<skip-directive>          ::= !SKIP  [<letter>]  ;      (8.1)

<begin-directive>         ::= !BEGIN [<letter>]  ;      (8.1)

<end-directive>           ::= !END  ;
```

Semantics:

<Skip-directives> are used in conjunction with the following <begin-directive> - <end-directive> pairs in the same module to cause text enclosed by such directive pairs to be ignored during compilation. A <skip-directive> with a <letter> activates all following pairs in which the <begin-directive> has the same <letter>; a <skip-directive> with no <letter> activates all following pairs, regardless of <letter>.

After an active <begin-directive> all symbols are ignored except nested !BEGIN/!END symbol pairs and the !END symbol which terminates suppression. Since the symbols are ignored, even <copy-directives> and <define-calls> are not recognized as such. The nested !BEGIN/!END symbol pairs have no effect other than pairing and neither these nor the terminating !END symbol are tested as to whether their context would be valid if the text were not being suppressed.

<Skip-directives> that do not activate any <begin-directive>, and <begin-directives> and <end-directives> not accounted for above, have no effect.

Constraints:

All <begin-directives> and <end-directives> in a <module> must be uniquely paired within that <module>.

9.3 LINKAGE DIRECTIVES

Syntax:

```
<linkage-directive>       ::= !LINKAGE
                              <symbol>...  ;          (8.2)
```

Semantics:

The <linkage-directive> indicates that the specified subroutine does not obey standard J73 linkage conventions. The <symbol> string specifies the implementation-dependent linkage type to be used in linking the procedure.

Constraints:

The <linkage-directive> must only occur in a <subroutine-declaration> between the heading and the declaration or in a <subroutine-definition> between the heading and the body.

If any <subroutine-declaration> or <subroutine-definition> contains a <linkage-directive>, then every declaration or definition for that same subroutine, whether in the same or different modules, must contain an equivalent <linkage-directive>.

9.4 TRACE DIRECTIVES

Syntax:

```
<trace-directive>      ::= !TRACE
                           [<trace-control>]
                           <name>,... ;                (8.2.1)

<trace-control>        ::= ( <boolean-formula> )       (5.2.2)
```

Semantics:

The <trace-directive> provides a run-time facility to trace program flow and to monitor data assignment. This "tracing" will be active from the lexical point at which the <trace-directive> occurs in the source until the end of the scope containing the directive. Its effect extends into nested subroutines declared within this lexical range of statements.

The <names> in the <trace-directive> are the names that will be traced, i.e., certain uses of these names as described in the following sentences will be noted in an implementation-dependent manner, for example on a symbolic output file. For <statement-names>, tracing of the associated statement will be noted each time the statement is fallen into or branched to. For <data-names>, modification of the data object and its new value will be noted. Modification of a data object is considered to have occurred upon execution of an <assignment-statement> in which the data object is the target or upon return from a subroutine to which the data object was passed as an <actual-output-parameter>. For <table-names>, modification of the entire table, a <table-entry>, or an item in the table will be noted. For <block-names>, modification of any data contained in the block will be noted. For <subroutine-names>, each call to the subroutine will be noted. If the subroutine containing the <trace-directive> is named in the directive and the directive is placed immediately after the <procedure-heading> or <function-heading>, entry and exit to that subroutine will be noted.

If a <trace-control> appears in the <trace-directive>, the <trace-control> formula will be tested dynamically at each use of a <name> as defined in the preceding paragraph. The trace output is suppressed if the formula is determined to be false. If the <trace-control> is omitted, it is considered to be true.

If two or more active <trace-directives> contain the same <name>, then the lexically latest one overrides the earlier ones for that <name>.

Constraints:

All <names> in the <trace-directive>, including names used in the <trace-control>, except for <statement-names> and <subroutine-names>, must have been declared prior to their use in the <trace-directive>.

A <bit-formula> cannot be implicitly converted to the <boolean-formula> in a <trace-control>.

9.5 INTERFERENCE DIRECTIVES

Syntax:

```
<interference-directive>      ::= !INTERFERENCE
                                  <interference-control>   ;

<interference-control>        ::= <data-name>  :            (2.6)
                                  <data-name>,...           (2.6)
```

Semantics:

The <interference-directive> informs the compiler that it cannot assume that the storage associated with the name to the left of the colon is distinct from the storage associated with the names to the right of the colon. In the absence of an <interference-directive> the compiler can make optimizations on the assumption that distinct <data-names> refer to distinct storage locations. If two <data-names> refer to the same storage location, these optimizations could result in erroneous code. If two <data-names> share the same storage, an assignment to one name should affect the value of the other. If the compiler optimizes on the assumption of non-interference, these semantics might not be preserved.

The compiler is aware of storage overlapping as a result of <specified-table-items> and as a result of the arrangement of data within a single overlay. This overlapping need not be reported via an <interference-directive>. Other instances of overlap, e.g., as a result of absolute addresses in separate overlays, must be stated by the use of an <interference-directive>.

Constraints:

An <interference-directive> must not follow a <statement>.

All <data-names> in the <interference-control> must have been declared prior to their use in the <interference-directive>.

9.6 REDUCIBLE DIRECTIVES

Syntax:

<reducible-directive> ::= !REDUCIBLE ;

Semantics:

The <reducible-directive> is used to allow additional optimization of function-calls. A reducible function is one for which all calls with identically-valued actual parameters result in identical function values and output parameter values, and which does not modify any data except actual output parameters and automatic data declared within its own body. If a <reducible-directive> is used to designate such functions as reducible, the compiler may detect the existence of such common calls, save the values returned from the initial call for use in place of any subsequent calls, and delete these subsequent calls.

Constraints:

The <reducible-directive> must only occur in a <function-declaration> between the heading and the declaration or in a <function-definition> between the heading and the body.

If any <function-declaration> contains a <reducible-directive>, then the <function-definition> for that function, whether in the same or different modules, must contain a <reducible-directive>.

9.7 LISTING DIRECTIVES

9.7.1 SOURCE-LISTING DIRECTIVES

Syntax:

<nolist-directive> ::= !NOLIST ;

<list-directive> ::= !LIST ;

<eject-directive> ::= !EJECT ;

Semantics:

Listing directives are used to provide source listing control information to the compiler. The <nolist-directive> causes suppression of the source listing beginning with the next source line, up to and including the next <list-directive>, which causes the listing to be resumed.

The <eject-directive> causes a page eject of the source listing before listing the following source lines. The <eject-directive> is ignored if the source listing is suppressed.

9.7.2 DEFINE-LISTING DIRECTIVES

Syntax:

```
<listinv-directive>          ::= !LISTINV  ;

<listexp-directive>          ::= !LISTEXP  ;

<listboth-directive>         ::= !LISTBOTH  ;
```

Semantics:

Define-listing directives allow programmer control over the text to be included in the source program listing for <define-calls>.

The text contained in the listing for a particular <define-call> depends on the define-listing directive which was in effect at the point of the corresponding <define-declaration> (not on the directive in effect at the point of the <define-call>). If this directive was !LISTINV, then the listing contains the text of the <define-call>; if the directive was !LISTEXP, then the listing contains the expanded string (the <define-string> after substitution of <actual-define-parameters>); if the directive was !LISTBOTH, then the listing contains both the invocation and the expansion.

Each define-listing directive is in effect from the lexical point at which it appears to the end of the current scope or to the point at which the next define-listing directive appears, whichever is first. The default define-listing directive in effect at the beginning of every module is !LISTINV.

Constraint:

<Listinv-directives>, <listexp-directives> and <listboth-directives> must not follow a <statement>.

Note:

The effect of a define-listing directive for a particular <define-call> is independent of whether a <nolist-directive> is suppressing the source listing at the point of the <define-declaration> being invoked.

9.8 REGISTER DIRECTIVES

Syntax:

```
<base-directive>             ::= !BASE <data-name>                (2.6)
                                 <integer-literal>  ;             (8.3.1)

<isbase-directive>           ::= !ISBASE <data-name>              (2.6)
                                 <integer-literal>  ;             (8.3.1)

<drop-directive>             ::= !DROP
                                 <integer-literal>  ;             (8.3.1)
```

Semantics:

Register directives affect target-machine register allocation. Each of these three directives uses an <integer-literal> in a target-machine-dependent way to specify which register is affected.

The <base-directive> loads the specified register with the address of the object corresponding to the <data-name>.

The <isbase-directive> directs the compiler to assume that the specified register contains the address of the data object corresponding to the <data-name>, but to take no action to guarantee it.

The <drop-directive> frees the specified register for other use by the compiler in generating code for subsequent statements. Both !BASE and !ISBASE cause the compiler to dedicate the register to the value it currently contains until !DROP or the end of the current scope is encountered.

Register directives may be ignored in implementations for machines on which register allocation is not meaningful.

9.9 EXPRESSION EVALUATION ORDER DIRECTIVES

Syntax:

```
<leftright-directive>          ::= !LEFTRIGHT  ;

<rearrange-directive>          ::= !REARRANGE  ;
```

Semantics:

If a <leftright-directive> is in effect, operators at the same precedence level are evaluated in left-to-right order within a given <formula>, consistent with the order imposed by parentheses.

If a <rearrange-directive> is in effect, order of evaluation is still constrained by parentheses and operator precedence, but the compiler is otherwise free to rearrange the expression for more optimal code generation, such as by applying associative and commutative laws.

The effect of each of these directives extends from the point at which it appears to the end of the current namescope or to the point at which a different expression-evaluation-order directive appears, whichever is first. At the beginning of each module, a <rearrange-directive> is in effect by default.

9.10 INITIALIZATION DIRECTIVES

Syntax:

<initialize-directive> ::= !INITIALIZE ;

Semantics:

The <initialize-directive> causes all STATIC data objects that are not explicitly initialized via an <item-preset>, <table-preset>, or <block-preset>, to be preset by default to all zero bits.

Its effect extends from the point at which it appears to the end of the current namescope.

Constraint:

The <initialize-directive> must not follow a <statement>, and may not appear in a table body nor in a <block-body-part> nor in a <subroutine-declaration>.

9.11 ALLOCATION ORDER DIRECTIVES

Syntax:

<order-directive> ::= !ORDER ;

Semantics:

The <order-directive> directs a compiler to allocate storage for the data objects in a block or ordinary table in the order in which their declarations appear in the text of the <block-body-part> or the <ordinary-table-body>. Lexically declared data objects that occur earlier in text are allocated physically lower addresses, and if data objects share a word, lexically earlier data are allocated to the left of later data. In the absence of an <order-directive>, a compiler is free to rearrange the physical storage layout for ease of access or more optimal utilization of memory.

The effect of the <order-directive> extends from the point at which it appears to the end of the current block or table. If the <order-directive> is in a block, its effect extends to the components of any blocks or ordinary tables contained in the block.

If an <order-directive> appears in an <ordinary-table-body> in a <table-type-declaration>, the ordering extends to all tables declared of that type.

Constraints:

A block affected by an <order-directive> cannot contain an <overlay-declaration>.

The <order-directive>, if present, must immediately follow the BEGIN token in a <block-body-part> or <ordinary-table-body>.

APPENDIX

CROSS-REFERENCE INDEX

This appendix provides a cross-reference for terminal and non-terminal constructs in the J73 syntax used in this manual. For each construct, columns give the section in the manual where it is defined and the sections where it is used or referenced.

Construct	Definition	References
A		2.1.1.3, 7.0, 8.1, 8.3.2
ABORT		4.5, 4.10, 8.2.2
abort-phrase	4.5	4.5
abort-statement	4.10	4.0
ABS		6.3.6, 8.2.2
abs-function	6.3.6	6.3
absolute-address	2.6	2.6
actual-define-parameter	2.4.1	2.4.1
actual-define-parameter-list	2.4.1	2.4.1
actual-input-parameter	4.5	4.5
actual-output-parameter	4.5	4.5
actual-parameter-list	4.5	4.5, 6.3
allocation-specifier	2.1.5	2.1.1, 2.1.2, 2.1.4
AND		5.2, 8.2.2, 8.2.3
and-continuation	5.2	5.2
arithmetic-operator	8.2.3	8.2.3
assignment-operator	8.2.3	8.2.3
assignment-statement	4.1	4.0
B		2.1.1.4, 7.0, 8.1, 8.3.2
BASE		9.8

Construct	Definition	References
base-directive	9.8	9.0
bead	8.3.2	8.3.2
bead-size	8.3.2	8.3.2
BEGIN		1.2.3, 2.0, 2.1.2.3, 2.1.2.4, 2.1.4, 2.5.1, 2.5.2, 2.7, 3.1, 4.0, 4.4, 8.2.2, 9.2.2
begin-directive	9.2.2	9.0
BIT		6.1, 8.2.2
bit-conversion	7.0	5.2
bit-formula	5.2	4.4, 5.0, 5.2, 5.2.2, 6.3.3, 6.3.5
bit-function	6.3.3	6.3
bit-function-call	5.2	5.2
bit-function-variable	6.1	6.1
bit-item-description	2.1.1.4	2.1.1.4
bit-literal	8.3.2	5.2, 8.3
bit-operator	8.2.3	8.2.3
bit-primary	5.2	5.2, 5.2.1
BITSINBYTE		1.4
BITSINPOINTER		1.4
BITSINWORD		1.4
BITSIZE		6.3.8, 8.2.2
bit-size	2.1.1.4	2.1.1.4
bits-per-entry	2.1.2.2	2.1.2.2
bit-type-conversion	7.0	7.0

Construct	Definition	References
bit-type-description	2.1.1.4	2.1.1, 7.0
bit-type-name	2.1.1.4	2.1.1.4, 7.0
bit-variable	5.2	5.2, 6.1
BLOCK		2.1.4, 2.2, 8.2.2
block-body-options	2.1.4	2.1.4
block-body-part	2.1.4	2.1.4, 2.2
block-declaration	2.1.4	2.1
block-dereference	6.1	6.1
block-item	6.1	6.1
block-name	2.1.4	2.1.4, 2.5.1, 2.6, 6.1, 6.3.8
block-preset	2.1.6	2.1.4
block-preset-list	2.1.6	2.1.6
block-preset-values-option	2.1.6	2.1.6
block-reference	6.1	4.5, 6.3.1
block-table	6.1	6.1
block-table-entry	6.1	6.1
block-table-item	6.1	6.1
block-type-declaration	2.2	2.2
block-type-name	2.2	2.1.1.7, 2.1.4, 2.2
boolean-formula	5.2.2	4.2, 4.3, 9.4
boolean-literal	8.3.3	5.2, 8.3
bounds-argument	6.3.9	6.3.9
bounds-function	6.3.9	6.3
BY		4.2, 8.2.2

Construct	Definition	References
by-formula	4.2	4.2
by-or-then-phrase	4.2	4.2
by-phrase	4.2	4.2
BYREF		3.3, 8.2.2
BYRES		3.3, 8.2.2
BYTE		6.1, 6.3.4, 8.2.2
byte-function	6.3.4	6.3
byte-function-variable	6.1	6.1
BYTEPOS		1.4
BYTESINWORD		1.4
BYTESIZE		6.3.8, 8.2.2
BYVAL		3.3, 8.2.2
C		2.1.1.5, 7.0, 8.1, 8.3.2
CASE	4.4	4.4
case-alternative	4.4	4.4
case-body	4.4	4.4
case-index	4.4	4.4
case-index-group	4.4	4.4
case-selector-formula	4.4	4.4
case-statement	4.4	4.0
character	8.1	2.4, 2.4.1, 8.3.4, 8.4
character-conversion	7.0	5.3

Construct	Definition	References
character-formula	5.3	4.4, 5.0, 5.2.1, 5.3, 6.3.4
character-function-call	5.3	5.3
character-item-description	2.1.1.5	2.1.1.5
character-literal	8.3.4	8.3, 9.1, 9.2.1
character-size	2.1.1.5	2.1.1.5
character-type-description	2.1.1.5	2.1.1, 7.0
character-type-name	2.1.1.5	2.1.1.5, 7.0
character-variable	5.3	5.3, 6.1
comment	8.4	8.2
compile-time-bit-formula	5.2	4.4, 5.0
compile-time-character-formula	5.3	4.4, 5.0
compile-time-fixed-formula	5.1.3	5.1
compile-time-floating-formula	5.1.2	1.4, 5.1
compile-time-formula	5.0	2.1.6
compile-time-integer-formula	5.1.1	1.4, 2.1.1.1, 2.1.1.2, 2.1.1.3, 2.1.1.4, 2.1.1.5, 2.1.1.6, 2.1.2.1, 2.1.2.2, 2.1.2.4, 2.1.6, 2.6, 4.4, 5.1, 6.3.9
compile-time-numeric-formula	5.1	5.0
compile-time-pointer-formula	5.5	5.0
compile-time-status-formula	5.4	2.1.2.1, 2.1.6, 4.4, 5.0
complete-program	1.1	
COMPOOL		1.2.1, 8.2.2, 9.1
compool-declaration	2.0	1.2.1, 2.0

Construct	Definition	References
compool-declared-name	9.1	9.1
compool-directive	9.1	9.0
compool-directive-list	9.1	9.1
compool-file-name	9.1	9.1
compool-module	1.2.1	1.1
compool-name	1.2.1	1.2.1
compound-def	2.5.1	2.5.1
compound-ref	2.5.2	2.5.2
compound-statement	4.0	4.0
CONDITION		8.2.2
conditional-statement	4.3	4.3
CONSTANT		8.2.2
constant-declaration	2.1.3	2.0, 2.1
constant-index	2.1.6	2.1.6
constant-item-name	2.1.3	2.1.3, 6.1, 6.2
constant-table-item-name	6.2	6.2
constant-table-name	2.1.3	2.1.3, 6.2
continuation	4.2	4.2
control-clause	4.2	4.2
control-item	4.2	4.2
controlled-statement	4.2	4.2
control-letter	4.2	4.2, 6.2
control-variable	4.2	4.2
COPY		9.2.1
copy-directive	9.2.1	9.0

Construct	Definition	References
D		2.1.2.3, 8.1, 8.3.2
data-declaration	2.1	2.0, 2.1.4, 2.5.1, 2.5.2
data-name	2.6	2.6, 3.3, 9.5, 9.8
declaration	2.0	1.2.2, 1.2.3, 2.0, 3.1, 3.2
DEF		1.2.2, 2.5.1, 8.2.2
DEFAULT		4.4, 8.2.2
default-option	4.4	4.4
default-preset-sublist	2.1.6	2.1.6
default-sublist	2.1.1.6	2.1.1.6
def-block-instantiation	2.5.1	2.5.1
DEFINE		2.4, 8.2.2
define-call	2.4.1	8.2
define-declaration	2.4	2.0
define-name	2.4	2.4, 2.4.1
define-string	2.4	2.4, 8.2
definition-part	2.4	2.4
def-specification	2.5.1	2.5
def-specification-choice	2.5.1	2.5.1
dereference	6.1	6.1
dereference-operator	8.2.3	8.2.3
digit	8.1	8.1, 8.2.1, 8.3.1, 8.3.2
dimension	2.1.2.1	2.1.2.1
dimension-list	2.1.2.1	2.1.2, 2.1.3, 2.2

Construct	Definition	References
dimension-number	6.3.9	6.3.9
directive	9.0	
DROP		9.8
drop-directive	9.8	9.0
E		8.1, 8.3.2
EJECT		9.7.1
eject-directive	9.7.1	9.0
ELSE		4.3, 8.2.2
else-clause	4.3	4.3
ENCAPSULATION		8.2.2
END		1.2.3, 2.0, 2.1.2.3, 2.1.2.4, 2.1.4, 2.5.1, 2.5.2, 2.7, 3.1, 4.0, 4.4, 8.2.2, 9.2.2
end-directive	9.2.2	9.0
entry-size	2.1.2.4	2.1.2.4
entry-specifier	2.1.2	2.1.2, 2.2
equal-or-not-equal-operator	8.2.3	5.2.1, 8.2.3
EQV		5.2, 8.2.2, 8.2.3
eqv-continuation	5.2	5.2
EXIT		4.8, 8.2.2
exit-statement	4.8	4.0
exponent	8.3.1	8.3.1
EXPORTS		8.2.2
external-declaration	2.5	2.0

Construct	Definition	References
F		2.1.1.2, 7.0, 8.1, 8.3.2
FALLTHRU		4.4, 8.2.2
FALSE		8.2.2, 8.3.3
fbit	6.3.3	6.1, 6.3.3
fbyte	6.3.4	6.1, 6.3.4
FIRST		6.3.11, 8.2.2
fixed-conversion	7.0	5.1.3
fixed-factor	5.1.3	5.1.3
fixed-formula	5.1.3	5.1, 5.1.3, 5.2.1
fixed-function-call	5.1.3	5.1.3
fixed-item-description	2.1.1.3	2.1.1.3
fixed-literal	8.3.1	5.1.3, 8.3.1
fixed-machine-parameter	1.4	5.1.3
FIXEDPRECISION		1.4
fixed-term	5.1.3	5.1.3
fixed-type-description	2.1.1.3	2.1.1, 7.0
fixed-type-name	2.1.1.3	2.1.1.3, 7.0
fixed-variable	5.1.3	5.1.3
floating-conversion	7.0	5.1.2
floating-factor	5.1.2	5.1.2
floating-formula	5.1.2	5.1, 5.1.2, 5.2.1
floating-function-call	5.1.2	5.1.2
floating-item-description	2.1.1.2	2.1.1.2
floating-literal	8.3.1	5.1.2, 8.3.1

Construct	Definition	References
floating-machine-parameter	1.4	5.1.2
floating-primary	5.1.2	5.1.2
floating-term	5.1.2	5.1.2
floating-type-description	2.1.1.2	2.1.1, 7.0
floating-type-name	2.1.1.2	2.1.1.2, 7.0
floating-variable	5.1.2	5.1.2
FLOATPRECISION		1.4
FLOATRADIX		1.4
FLOATRELPRECISION		1.4
FLOATUNDERFLOW		1.4
FOR		4.2, 8.2.2
for-clause	4.2	4.2
formal-define-parameter	2.4	2.4
formal-define-parameter-list	2.4	2.4
formal-input-parameter	3.3	3.3
formal-output-parameter	3.3	3.3
formal-parameter-list	3.3	3.1, 3.2
formula	5.0	4.1, 4.2, 4.5, 5.1.1, 5.1.2, 5.1.3, 5.2, 5.3, 5.4, 5.5, 5.6, 6.3.8
fractional-form	8.3.1	8.3.1
fraction-specifier	2.1.1.3	1.4, 2.1.1.3
FREE		8.2.2
function-body	3.2	3.2

Construct	Definition	References
function-call	6.3	5.1.1, 5.1.2, 5.1.3, 5.2, 5.3, 5.4
function-declaration	3.2	3.0
function-definition	3.2	3.0
function-heading	3.2	3.2
function-name	3.2	3.2, 3.3, 4.5, 6.1, 6.3, 6.3.1
G		8.1, 8.3.2
GOTO		4.7, 8.2.2
goto-statement	4.7	4.0
H		8.1, 8.3.2
HANDLER		8.2.2
I		8.1, 8.3.2
IF		4.3, 8.2.2
if-statement	4.3	4.0
IMPLFIXEDPRECISION		1.4
IMPLFLOATPRECISION		1.4
IMPLINTSIZE		1.4
IN		8.2.2
increment-amount	6.3.2	6.3.2
index	6.1	6.1
INITIALIZE		9.10
initialize-directive	9.10	9.0
initial-value	4.2	4.2
INLINE		3.4, 8.2.2

Construct	Definition	References
inline-declaration	3.4	2.0
input-parameter-name	3.3	3.3
INSTANCE		8.2.2
integer-conversion	7.0	5.1.1
integer-factor	5.1.1	5.1.1, 5.1.3
integer-formula	5.1.1	4.4, 4.9, 5.1, 5.1.1, 5.2.1, 6.1, 6.3.2, 6.3.3, 6.3.4, 6.3.5
integer-function-call	5.1.1	5.1.1
integer-item-description	2.1.1.1	2.1.1.1
integer-literal	8.3.1	5.1.1, 8.3.1, 9.8
integer-machine-parameter	1.4	5.1.1
integer-primary	5.1.1	5.1.1, 5.1.2
integer-size	2.1.1.1	1.4, 2.1.1.1
integer-term	5.1.1	5.1.1, 5.1.3
integer-type-description	2.1.1.1	2.1.1, 7.0
integer-type-name	2.1.1.1	2.1.1.1, 7.0
integer-variable	5.1.1	5.1.1
INTERFERENCE		9.5
interference-control	9.5	9.5
interference-directive	9.5	9.0
INTERRUPT		8.2.2
INTPRECISION		1.4
intrinsic-function-call	6.3	6.3
ISBASE		9.8
isbase-directive	9.8	9.0

Construct	Definition	References
ITEM		2.1.1, 2.1.2.3, 2.1.2.4, 8.2.2,
item	6.1	6.1
item-declaration	2.1.1	2.1
item-dereference	6.1	6.1
item-name	2.1.1	2.1.1, 2.6, 4.2, 6.1
item-preset	2.1.6	2.1.1, 2.1.3
item-preset-value	2.1.6	2.1.6
item-type-declaration	2.2	2.2
item-type-description	2.1.1	2.1.1, 2.1.2.3, 2.1.2.4, 2.2, 3.2
item-type-name	2.2	2.1.1.1, 2.1.1.2, 2.1.1.3, 2.1.1.4, 2.1.1.5, 2.1.1.6, 2.1.1.7, 2.2
J		8.1, 8.3.2
K		8.1, 8.3.2
L		8.1, 8.3.2
LABEL		2.3, 8.2.2
label	4.0	1.2.3, 3.1, 4.0, 4.4
LAST		6.3.11, 8.2.2
LBOUND		6.3.9, 8.2.2
LEFTRIGHT		9.9
leftright-directive	9.9	9.0
letter	8.1	2.1.1.6, 2.4, 4.2, 8.1, 8.2, 8.2.1

Construct	Definition	References
letter-digit-$-or-prime	8.2.1	8.2.1
letter-or-$	8.2.1	8.2.1
LIKE		2.2, 8.2.2
like-option	2.2	2.2
LINKAGE		9.3
linkage-directive	9.3	9.0
LIST		9.7.1
LISTBOTH		9.7.2
listboth-directive	9.7.2	9.0
list-directive	9.7.1	9.0
LISTEXP		9.7.2
listexp-directive	9.7.2	9.0
LISTINV		9.7.2
listinv-directive	9.7.2	9.0
literal	8.3	8.2
LOC		6.3.1, 8.2.2
loc-argument	6.3.1	6.3.1
location-specifier	2.1.2.4	2.1.2.4
loc-function	6.3.1	6.3
LOCSINWORD		1.4
logical-continuation	5.2	5.2
logical-operand	5.2	5.2
logical-operator	8.2.3	8.2.3
loop-statement	4.2	4.0

Construct	Definition	References
loop-type	4.2	4.2
lower-bound	2.1.2.1	2.1.2.1, 4.4
lower-bound-option	2.1.2.1	2.1.2.1
M		2.1.2.3, 8.1, 8.3.2
machine-specific-function-call	6.3	6.3
machine-specific-procedure-call	4.5	4.5
main-program-module	1.2.3	1.1
mark	8.1	8.1
MAXBITS		1.4
MAXBYTES		1.4
MAXFIXED		1.4
MAXFIXEDPRECISION		1.4
MAXFLOAT		1.4
MAXFLOATPRECISION		1.4
MAXINT		1.4
MAXINTSIZE		1.4
MAXSIGDIGITS		1.4
MAXTABLESIZE		1.4
MINFIXED		1.4
MINFLOAT		1.4
MINFRACTION		1.4
MININT		1.4
MINRELPRECISION		1.4
MINSCALE		1.4
MINSIZE		1.4

Construct	Definition	References
MOD		8.2.2, 8.2.3
module	1.1	1.1
multiply-divide-or-mod	8.2.3	5.1.1, 8.2.3
multiply-or-divide	8.2.3	5.1.2, 5.1.3, 8.2.3
N		2.1.2.3, 8.1, 8.3.2
name	8.2.1	1.2.1, 1.2.3, 2.1.1, 2.1.1.6, 2.1.2.3, 2.1.3, 2.2, 2.4, 3.1, 3.2, 4.0, 8.2, 9.1, 9.4
named-bit-constant	5.2	5.2
named-character-constant	5.3	5.3
named-constant	6.2	5.1.1, 5.1.2, 5.1.3, 5.2, 5.3, 5.4, 5.5, 5.6
named-fixed-constant	5.1.3	5.1.3
named-floating-constant	5.1.2	5.1.2
named-integer-constant	5.1.1	5.1.1
named-pointer-constant	5.5	5.5
named-status-constant	5.4	5.4
named-table-constant	5.6	5.6
named-variable	6.1	6.1, 6.3.1
nbit	6.3.3	6.1, 6.3.3
nbyte	6.3.4	6.1, 6.3.4
NENT		6.3.12, 8.2.2
nent-argument	6.3.12	6.3.12
nent-function	6.3.12	6.3

Construct	Definition	References
NEW		8.2.2
NEXT		6.3.2, 8.2.2
next-argument	6.3.2	6.3.2
next-function	6.3.2	6.3
NOLIST		9.7.1
nolist-directive	9.7.1	9.0
non-nested-subroutine	1.2.2	1.2.2, 1.2.3
NOT		5.2, 8.2.2, 8.2.3
NULL		8.2.2, 8.3.5
null-declaration	2.7	2.0, 2.1.2.3, 2.1.2.4, 2.1.4, 2.5.1, 2.5.2
null-statement	4.0	4.0
number	8.3.1	8.3.1
numeric-formula	5.1	4.2, 5.0, 6.3.6, 6.3.7
numeric-literal	8.3.1	8.3
NWDSEN		6.3.10, 8.2.2
nwdsen-argument	6.3.10	6.3.10
nwdsen-function	6.3.10	6.3
O		8.1, 8.3.2
operator	8.2.3	8.2
OR		5.2, 8.2.2, 8.2.3
or-continuation	5.2	5.2
ORDER		9.11
order-directive	9.11	9.0

Construct	Definition	References
ordinary-entry-specifier	2.1.2.3	2.1.2
ordinary-table-body	2.1.2.3	2.1.2.3
ordinary-table-item-declaration	2.1.2.3	2.1.2.3
ordinary-table-options	2.1.2.3	2.1.2.3
other-character		8.1
output-parameter-name	3.3	3.3
OVERLAY		2.6, 8.2.2
overlay-address	2.6	2.6
overlay-declaration	2.6	2.0, 2.1.4
overlay-element	2.6	2.6
overlay-expression	2.6	2.6
overlay-string	2.6	2.6
P		2.1.1.7, 7.0, 8.1, 8.3.2
packing-specifier	2.1.2.3	2.1.2.3
PARALLEL		2.1.2.2, 8.2.2
parameter-binding	3.3	3.3
plus-or-minus	8.2.3	5.1.1, 5.1.2, 5.1.3, 8.2.3
pointer-conversion	7.0	5.5
pointer-formula	5.5	5.0, 5.2.1, 5.5, 6.1, 6.3.2
pointer-function-call	5.5	5.5
pointer-item-description	2.1.1.7	2.1.1.7
pointer-item-name	6.1	6.1
pointer-literal	8.3.5	5.5

Construct	Definition	References
pointer-type-description	2.1.1.7	2.1.1, 7.0
pointer-type-name	2.1.1.7	2.1.1.7, 7.0
pointer-variable	5.5	5.5
POS		2.1.2.4, 2.1.6, 2.6
precision	2.1.1.2	1.4, 2.1.1.2
preset-index-specifier	2.1.6	2.1.6
preset-values-option	2.1.6	2.1.6
PROC		3.1, 3.2, 8.2.2
procedure-body	3.1	3.1
procedure-call-statement	4.5	4.0
procedure-declaration	3.1	3.0
procedure-definition	3.1	3.0
procedure-heading	3.1	3.1
procedure-module	1.2.2	1.1
procedure-name	3.1	3.1, 3.3, 4.5, 6.3.1
PROGRAM		1.2.3, 8.2.2
program-body	1.2.3	1.2.3
program-name	1.2.3	1.2.3
PROTECTED		8.2.2
Q		8.1, 8.3.2
R		2.1.1.2, 8.1, 8.3.2
READONLY		8.2.2
real-literal	8.3.1	8.3.1
REARRANGE		9.9

Construct	Definition	References
rearrange-directive	9.9	9.0
REC		3.1, 8.2.2
REDUCIBLE		9.6
reducible-directive	9.6	9.0
REF		2.5.2, 8.2.2
ref-specification	2.5.2	2.5
ref-specification-choice	2.5.2	2.5.2
REGISTER		8.2.2
relational-expression	5.2.1	5.2
relational-operator	8.2.3	5.2.1, 8.2.3
RENT		3.1, 8.2.2
REP		7.0, 8.2.2
rep-conversion	7.0	6.1, 7.0
repetition-count	2.1.6	2.1.6
rep-function-variable	6.1	6.1
reserved-word	8.2.2	2.1.1.6, 8.2
RETURN		4.6, 8.2.2
return-statement	4.6	4.0
round-or-truncate	2.1.1.2	2.1.1.1, 2.1.1.2, 2.1.1.3
S		2.1.1.1, 7.0, 8.1, 8.3.2
scale-specifier	2.1.1.3	1.4, 2.1.1.3
separator	8.2.4	8.2
SGN		6.3.7, 8.2.2

Construct	Definition	References
shift-count	6.3.5	6.3.5
shift-direction	6.3.5	6.3.5
shift-function	6.3.5	6.3
SHIFTL		6.3.5, 8.2.2
SHIFTR		6.3.5, 8.2.2
sign	8.3.1	5.1.1, 5.1.2, 5.1.3, 8.3.1
SIGNAL		8.2.2
sign-function	6.3.7	6.3
simple-def	2.5.1	2.5.1
simple-ref	2.5.2	2.5.2
simple-statement	4.0	4.0
size-argument	6.3.8	6.3.8
size-function	6.3.8	6.3
size-type	6.3.8	6.3.8
SKIP		9.2.2
skip-directive	9.2.2	9.0
spacer	2.6	2.6
specified-entry-specifier	2.1.2.4	2.1.2
specified-item-description	2.1.2.4	2.1.2.4
specified-preset-sublist	2.1.6	2.1.6
specified-sublist	2.1.1.6	2.1.1.6
specified-table-body	2.1.2.4	2.1.2.4
specified-table-item-declaration	2.1.2.4	2.1.2.4
specified-table-options	2.1.2.4	2.1.2.4

Construct	Definition	References
START		1.2.1, 1.2.2, 1.2.3, 8.2.2
starting-bit	2.1.2.4	2.1.2.4
starting-word	2.1.2.4	2.1.2.4
statement	4.0	1.2.3, 3.1, 4.0, 4.2, 4.3, 4.4
statement-name	4.0	2.3, 3.3, 4.0, 4.5, 4.7, 6.3.1
statement-name-declaration	2.3	2.0, 2.5.1
STATIC		2.1.5, 8.2.2
STATUS		2.1.1.6, 8.2.2
status	2.1.1.6	2.1.1.6
status-constant	2.1.1.6	2.1.1.6, 5.4, 8.2
status-conversion	7.0	5.4
status-formula	5.4	4.4, 5.0, 5.4, 6.1, 6.3.11
status-function-call	5.4	5.4
status-inverse-argument	6.3.11	6.3.11
status-inverse-function	6.3.11	6.3
status-item-description	2.1.1.6	2.1.1.6
status-list	2.1.1.6	2.1.1.6
status-list-index	2.1.1.6	2.1.1.6
status-size	2.1.1.6	2.1.1.6
status-type-description	2.1.1.6	2.1.1, 7.0
status-type-name	2.1.1.6	2.1.1.6, 6.3.11, 7.0
status-variable	5.4	5.4
STOP		4.9, 8.2.2

Construct	Definition	References
stop-statement	4.9	4.0
structure-specifier	2.1.2.2	2.1.2, 2.2
subroutine-attribute	3.1	3.1, 3.2
subroutine-body	3.1	3.1, 3.2
subroutine-declaration	3.0	2.0, 2.5.2
subroutine-definition	3.0	1.2.2, 1.2.3, 3.1
subroutine-name	3.3	3.3, 3.4
subscript	6.1	6.1, 6.2
symbol	8.2	9.3
T		2.1.1.2, 2.1.2.2, 8.1, 8.3.2
TABLE		2.1.2, 2.2, 8.2.2
table	6.1	6.1
table-conversion	7.0	5.6
table-declaration	2.1.2	2.1
table-dereference	6.1	6.1
table-description	2.1.2	2.1.2, 2.1.3
table-entry	6.1	6.1
table-formula	5.6	5.0, 5.6
table-item	6.1	6.1
table-item-name	2.1.2.3	2.1.2.3, 2.1.2.4, 6.1, 6.2
table-name	2.1.2	2.1.2, 2.6, 6.1, 6.3.9, 6.3.10, 6.3.12

Construct	Definition	References
table-preset	2.1.6	2.1.2, 2.1.2.3, 2.1.2.4
table-preset-list	2.1.6	2.1.6
table-type-declaration	2.2	2.2
table-type-name	2.2	2.1.1.7, 2.1.2, 2.2,6.3.9, 6.3.10, 6.3.12, 7.0
table-type-specifier	2.2	2.2
table-variable	5.6	5.6
TERM		1.2.1, 1.2.3, 8.2.2
THEN		4.2, 8.2.2
then-phrase	4.2	4.2
TO		8.2.2
TRACE		9.4
trace-control	9.4	9.4
trace-directive	9.4	9.0
TRUE		8.2.2, 8.3.3
TYPE		2.2, 8.2.2
type-declaration	2.2	2.0
type-name	2.1.1.7	2.1.1.7
U		2.1.1.1, 7.0, 8.1, 8.3.2
UBOUND		6.3.9, 8.2.2
UPDATE		8.2.2
upper-bound	2.1.2.1	2.1.2.1, 4.4
user-defined-function-call	6.3	6.3
user-defined-procedure-call	4.5	4.5

Construct	Definition	References
V		2.1.1.6, 2.1.2.4, 8.1, 8.3.2
variable	6.1	4.1, 4.5, 5.1.1, 5.1.2, 5.1.3, 5.2, 5.3, 5.4, 5.5, 5.6
variable-list	4.1	4.1
W		2.1.2.4, 2.6, 8.1
which-bound	6.3.9	6.3.9
WHILE		4.2, 8.2.2
while-clause	4.2	4.2
while-phrase	4.2	4.2
WITH		8.2.2
WORDSIZE		6.3.8, 8.2.2
words-per-entry	2.1.2.4	2.1.2.4
WRITEONLY		8.2.2
X		8.1
XOR		5.2, 8.2.2, 8.2.3
xor-continuation	5.2	5.2
Y		8.1
Z		2.1.1.2, 8.1
ZONE		8.2.2

Custodians:

Air Force - 02

Preparing Activity:

Air Force - 17
(Project IPSC0210)

☆U.S. GOVERNMENT PRINTING OFFICE: 1984—705-040/A3613